MLOps

Notes de Cours

Master M2 — 2025–2026

Yaë Ulrich Gaba

« L'intelligence artificielle est la nouvelle électricité. »
— Andrew Ng

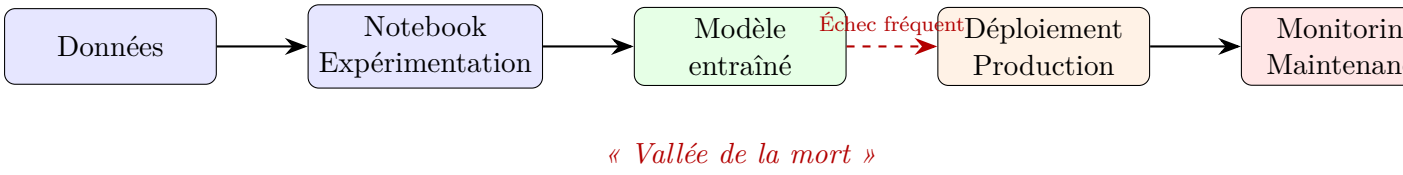
25 mars 2026

Table des matières

Préface

Pourquoi 90 % des projets ML échouent en production

Un constat récurrent dans l'industrie est alarmant : selon Gartner (2022), seulement 54 % des modèles de machine learning (ML) passent du prototype à la production, et parmi ceux-ci, beaucoup sont retirés dans les premiers mois. VentureBeat rapportait en 2019 que 87 % des projets de data science n'atteignent jamais la production. Ce n'est pas un problème de modélisation — c'est un problème d'**ingénierie**.



Les causes principales de ces échecs sont :

- **Dette technique** : le code ML accumule de la complexité cachée (dépendances de données, boucles de rétroaction, configurations non versionnées) [10].
- **Irréproductibilité** : impossible de recréer les résultats d'un collègue, même avec le même code, car l'environnement, les données ou les hyperparammètres diffèrent.
- **Fossé organisationnel** : les data scientists travaillent en notebooks isolés, les ingénieurs ML ne comprennent pas les choix de modélisation, et les équipes ops ne savent pas surveiller un modèle.
- **Absence de monitoring** : un modèle déployé sans surveillance dérive silencieusement (*data drift*, *concept drift*).
- **Pas de pipeline reproductible** : chaque étape (prétraitement, entraînement, évaluation) est exécutée manuellement.

Ce que le MLOps résout

Le **MLOps** (Machine Learning Operations) est un ensemble de pratiques qui unifient le développement ML (Dev) et les opérations (Ops) pour automatiser, standardiser et fiabiliser le cycle de vie des modèles. C'est l'équivalent du DevOps pour le machine learning, avec des défis spécifiques : versionnement des données, suivi des expériences, reproductibilité des résultats, monitoring de la dérive.

Organisation de ce cours

Ce cours est conçu pour des étudiants de niveau Master en science des données, apprentissage automatique ou mathématiques appliquées. Il suppose des bases solides en Python, une familiarité avec la ligne de commande Linux/Unix, et des connaissances en apprentissage automatique.

Chaque chapitre suit une structure pédagogique cohérente :

1. **Motivation et concepts** : pourquoi cet outil ou cette pratique est nécessaire.
2. **Tutoriel pratique** : code fonctionnel que l'étudiant peut exécuter sur sa machine.
3. **Bonnes pratiques et anti-patterns** : ce qu'il faut faire et ce qu'il faut éviter.
4. **Comparaison d'outils** : tableaux comparatifs quand plusieurs solutions existent.
5. **Mini-projet** : exercice intégrateur utilisant les outils des chapitres précédents.
6. **Exercices** : gradués par difficulté (★ configuration, ★★ construction de pipeline, ★★★ projet de bout en bout).
7. **Aide-mémoire** : résumé des commandes essentielles.

Remarque terminologique. Le MLOps emprunte massivement au vocabulaire anglophone. Quand un terme technique n'a pas d'équivalent français établi (par exemple *feature store*, *drift*, *pipeline*), nous l'utilisons en anglais, en italique lors de sa première apparition, avec une explication en français.

Bonne lecture et bons déploiements !

Chapitre 1

Introduction au MLOps — Du Notebook à la Production

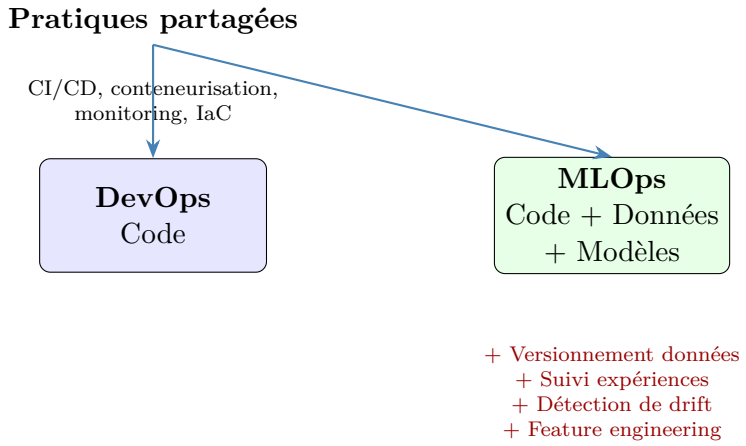
1.1 Qu'est-ce que le MLOps ?

Le **MLOps** désigne l'ensemble des pratiques, outils et principes culturels qui visent à déployer et maintenir des modèles de machine learning en production de manière fiable et efficace. Le terme est un mot-valise combinant *Machine Learning* et *Operations*, par analogie avec le *DevOps* du génie logiciel.

Définition 1.1 (MLOps)**.** Le MLOps est la discipline qui applique les principes du DevOps (intégration continue, déploiement continu, monitoring, automatisation) au cycle de vie des modèles de machine learning, en y ajoutant les spécificités liées aux données, aux expériences et à la dérive des modèles.

1.1.1 DevOps vs MLOps

Le DevOps traditionnel gère du code. Le MLOps gère du code **et** des données **et** des modèles — trois artefacts dont l'interaction crée une complexité supplémentaire.

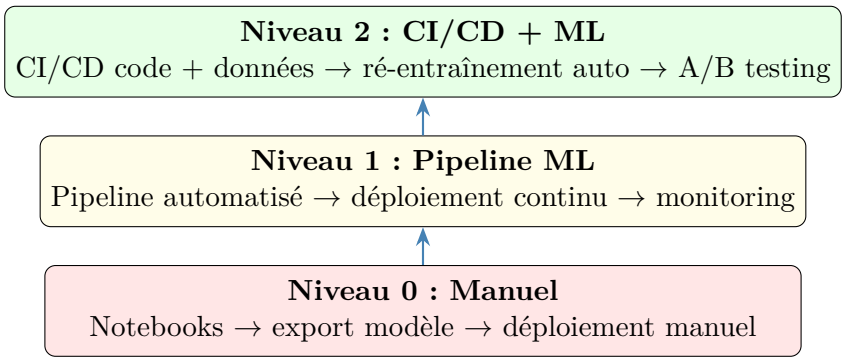


Remarque 1.2. La différence fondamentale est que le comportement d'un système ML dépend non seulement du code mais aussi des données. Un même code entraîné sur des données différentes produit un modèle différent. Cela nécessite de versionner les données au même titre que le code.

1.2 Les niveaux de maturité MLOps

Google a proposé trois niveaux de maturité pour les systèmes ML en production :

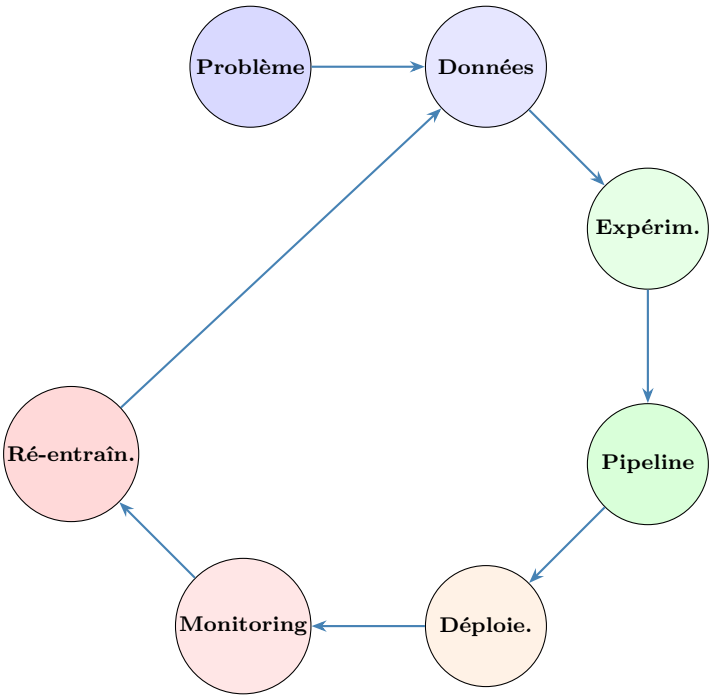
Niveau	Nom	Description
0	Manuel	Processus entièrement manuel. Notebooks, pas de pipeline. Séparation complète entre data scientists et ops.
1	Pipeline ML	Pipeline automatisé d'entraînement. Déploiement continu du modèle. Monitoring basique.
2	Pipeline CI/CD	Pipeline CI/CD pour le code ML. Entraînement automatisé déclenché par les données. Ré-entraînement automatique en cas de drift.



1.3 Le cycle de vie ML

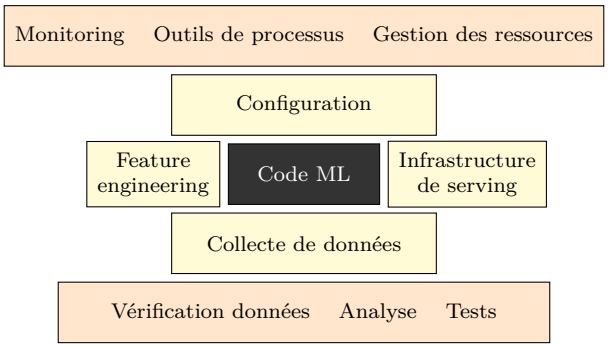
Un projet ML en production traverse plusieurs phases, qui forment un cycle (et non une séquence linéaire) :

1. **Définition du problème** : quel est l'objectif métier ? Quel est le critère de succès ?
2. **Collecte et préparation des données** : acquisition, nettoyage, labellisation, feature engineering.
3. **Expérimentation** : choix du modèle, optimisation des hyperparamètres, évaluation.
4. **Développement du pipeline** : transformation du notebook en code modulaire, testé et versionné.
5. **Déploiement** : mise en production du modèle (API, batch, edge).
6. **Monitoring** : surveillance de la performance, détection de drift, alertes.
7. **Ré-entraînement** : mise à jour du modèle quand les données ou les performances évoluent.



1.4 La dette technique en ML

Sculley et al. [10] ont identifié que le code ML ne représente qu'une petite fraction d'un système ML en production. Le reste est de l'infrastructure :



Le piège du notebook

Le notebook Jupyter est un excellent outil d'exploration, mais un piètre outil de production. Les problèmes classiques :

- Exécution non linéaire (cellules exécutées dans le désordre)
- État caché (variables globales qui persistent)
- Pas de tests unitaires
- Difficulté de versionnement (fichiers JSON volumineux)
- Pas de modularité (fonctions et classes non réutilisables)

1.5 Architecture d'un système MLOps

Un système MLOps mature comprend les composants suivants :

Composants d'un système MLOps

Gestion de code Git, GitHub/GitLab

Gestion de données DVC, Delta Lake, LakeFS

Gestion d'environnements Conda, pip, venv, Poetry

Suivi d'expériences MLflow, Weights & Biases, Neptune

Pipeline d'entraînement Airflow, Prefect, Kubeflow Pipelines

Feature store Feast, Tecton, Hopsworks

Conteneurisation Docker, Kubernetes

Serving FastAPI, TorchServe, TensorFlow Serving, Triton

CI/CD GitHub Actions, GitLab CI, Jenkins

Monitoring Prometheus, Grafana, Evidently, WhyLabs

1.6 Tutoriel : du notebook au script modulaire

1.6.1 Le notebook de départ

Considérons un notebook typique de classification :

Notebook typique (anti-pattern)

```
# Cellule 1
import pandas as pd
from sklearn.ensemble import RandomForestClassifier
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score

# Cellule 2
df = pd.read_csv("data.csv")
X = df.drop("target", axis=1)
y = df["target"]

# Cellule 3
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.2, random_state=42
)

# Cellule 4
model = RandomForestClassifier(n_estimators=100, random_state=42)
```

```
model.fit(X_train, y_train)
print(f"Accuracy: {accuracy_score(y_test, model.predict(X_test)):.4f}")
```

1.6.2 Le script modulaire

Structure de projet recommandée

```
my_ml_project/
  src/
    __init__.py
    data.py          # Chargement et preprocessing
    model.py         # Definition du modele
    train.py         # Boucle d'entrainement
    evaluate.py      # Metriques d'evaluation
  tests/
    test_data.py
    test_model.py
  configs/
    config.yaml      # Hyperparametres
  requirements.txt
  Makefile
  README.md
```

src/data.py — Module de données

```python
"""Data loading and preprocessing module."""
import pandas as pd
from sklearn.model_selection import train_test_split


def load_data(path: str) -> pd.DataFrame:
    """Load dataset from CSV file."""
    df = pd.read_csv(path)
    return df


def split_data(
    df: pd.DataFrame,
    target_col: str = "target",
    test_size: float = 0.2,
    random_state: int = 42,
) -> tuple:
    """Split data into train and test sets."""
    X = df.drop(target_col, axis=1)
    y = df[target_col]
    return train_test_split(
        X, y, test_size=test_size, random_state=random_state
```

```
    )
```

src/train.py — Script d'entraînement

```python
"""Training script with configuration."""
import argparse
import yaml
import joblib
from data import load_data, split_data
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import accuracy_score


def train(config: dict) -> None:
    """Train model with given configuration."""
    # Data
    df = load_data(config["data"]["path"])
    X_train, X_test, y_train, y_test = split_data(
        df,
        target_col=config["data"]["target_col"],
        test_size=config["data"]["test_size"],
    )

    # Model
    model = RandomForestClassifier(**config["model"])
    model.fit(X_train, y_train)

    # Evaluate
    y_pred = model.predict(X_test)
    acc = accuracy_score(y_test, y_pred)
    print(f"Accuracy: {acc:.4f}")

    # Save
    joblib.dump(model, config["output"]["model_path"])
    print(f"Model saved to {config['output']['model_path']}")


if __name__ == "__main__":
    parser = argparse.ArgumentParser()
    parser.add_argument("--config", default="configs/config.yaml")
    args = parser.parse_args()

    with open(args.config) as f:
        config = yaml.safe_load(f)

    train(config)
```

configs/config.yaml

```yaml
data:
  path: "data/dataset.csv"
  target_col: "target"
  test_size: 0.2

model:
  n_estimators: 100
  max_depth: 10
  random_state: 42

output:
  model_path: "models/rf_model.joblib"
```

Makefile

```
.PHONY: install train test clean

install:
	pip install -r requirements.txt

train:
	python src/train.py --config configs/config.yaml

test:
	pytest tests/ -v

clean:
	rm -rf models/*.joblib __pycache__ .pytest_cache
```

1.7 Bonnes pratiques et anti-patterns

Principes fondamentaux

1. **Séparation des préoccupations** : données, modèle, entraînement, évaluation dans des modules distincts.
2. **Configuration externalisée** : hyperparammètres dans des fichiers YAML, pas en dur dans le code.
3. **Reproductibilité par défaut** : fixer les graines aléatoires, versionner les données et le code.
4. **Tests automatiques** : tester le chargement des données, la forme des tenseurs, la convergence sur un petit échantillon.
5. **Automatisation** : Makefile ou scripts pour toutes les opérations répétitives.

À éviter absolument

- Hardcoder des chemins absolus : `/home/alice/data/train.csv`
- Committer des données volumineuses dans Git
- Committer des secrets (clés API, mots de passe)
- Travailler exclusivement en notebooks sans code modulaire
- Ignorer les tests sous prétexte que « c'est du ML »
- Déployer un modèle sans monitoring

1.8 Mini-projet : structurer un projet ML

Mini-projet 1 : Du notebook au projet structuré

1. Créez la structure de répertoires recommandée.
2. Transformez le notebook de classification ci-dessus en modules Python séparés.
3. Écrivez un fichier `config.yaml` pour les hyperparammètres.
4. Créez un `Makefile` avec les cibles `install`, `train`, `test`, `clean`.
5. Écrivez au moins deux tests unitaires (un pour le chargement des données, un pour la forme de la sortie du modèle).
6. Vérifiez que `make train` fonctionne de bout en bout.

1.9 Exercices

Exercice 1.1 (★ — Structure de projet)**.** Créez un squelette de projet ML pour un problème de régression avec la structure recommandée. Incluez un `README.md` décrivant le projet et un `.gitignore` adapté au ML (ignorer `data/`, `models/`, `*.pyc`, `.ipynb_checkpoints/`).

Exercice 1.2 (★★ — Refactoring de notebook)**.** Prenez un notebook Jupyter existant (le vôtre ou un notebook Kaggle) et transformez-le en projet structuré :

1. Identifiez les fonctions réutilisables
2. Créez les modules Python correspondants
3. Externalisez la configuration
4. Écrivez des tests pour les fonctions critiques

Exercice 1.3 (★★★ — Projet complet)**.** Construisez un projet ML complet pour la classification de texte (par exemple, analyse de sentiment sur le dataset IMDb) :

1. Structure modulaire complète
2. Configuration YAML
3. Tests unitaires avec `pytest`
4. Script d'entraînement avec arguments en ligne de commande
5. Rapport automatique des métriques (précision, rappel, F1)
6. `Makefile` complet

1.10 Aide-mémoire

Résumé du chapitre 1

Concept	Point clé
MLOps	DevOps + données + modèles
Niveaux de maturité	0 (manuel) → 1 (pipeline) → 2 (CI/CD)
Dette technique	Le code ML est une petite fraction du système
Anti-pattern notebook	État caché, pas de tests, pas de modularité
Structure projet	`src/`, `tests/`, `configs/`, Makefile
Config externalisée	YAML pour les hyperparammètres

Chapitre 2

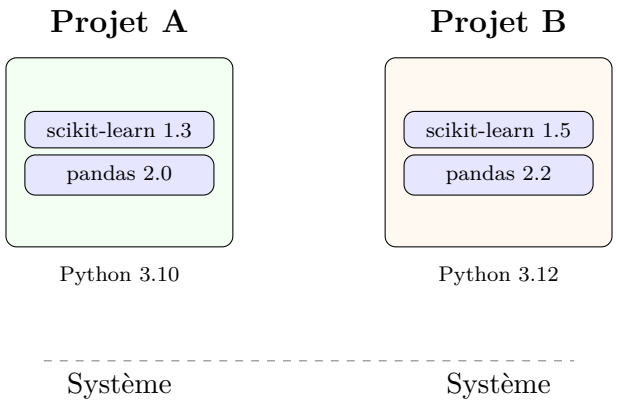
Environnements et Gestion de Dépendances

2.1 Pourquoi isoler les environnements ?

Un problème classique en ML : « Ça marche sur ma machine ! » (*works on my machine*). Les causes :

- Versions de Python différentes
- Versions de bibliothèques incompatibles
- Dépendances système manquantes (CUDA, cuDNN)
- Variables d'environnement non configurées

Définition 2.1 (Environnement virtuel)**.** Un **environnement virtuel** est un répertoire isolé contenant une installation Python spécifique et un ensemble de packages, indépendant du système et des autres projets.



2.2 venv — L'outil intégré à Python

`venv` est le module standard de Python pour créer des environnements virtuels. Il est simple, léger, et ne nécessite aucune installation supplémentaire.

Commandes venv essentielles

```
# Creer un environnement virtuel
python -m venv .venv

# Activer (Linux/macOS)
source .venv/bin/activate

# Activer (Windows)
.venv\Scripts\activate

# Installer des packages
pip install numpy pandas scikit-learn

# Sauvegarder les dependances
pip freeze > requirements.txt

# Recreer l'environnement
pip install -r requirements.txt

# Desactiver
deactivate
```

Pièges de pip freeze

`pip freeze` inclut **toutes** les dépendances transitives, ce qui rend le fichier fragile. Préférez :

- Maintenir manuellement un `requirements.txt` avec les dépendances directes et leurs versions.
- Utiliser `pip-tools` pour générer un `requirements.txt` verrouillé à partir d'un `requirements.in`.

2.2.1 pip-tools : le verrouillage des dépendances

Workflow pip-tools

```
# Installer pip-tools
pip install pip-tools

# requirements.in (dependances directes seulement)
# numpy>=1.24
# pandas>=2.0
# scikit-learn>=1.3

# Compiler (resoudre et verrouiller)
pip-compile requirements.in -o requirements.txt
```

```
# Installer exactement les versions verrouillees
pip-sync requirements.txt
```

2.3 Conda — Environnements avec dépendances système

Conda gère non seulement les packages Python mais aussi les bibliothèques système (CUDA, MKL, OpenSSL). C'est l'outil privilégié pour les projets ML nécessitant des GPU.

Commandes Conda essentielles

```
# Creer un environnement
conda create -n ml-project python=3.11

# Activer
conda activate ml-project

# Installer des packages
conda install numpy pandas scikit-learn
conda install -c pytorch pytorch torchvision  # canal PyTorch

# Exporter l'environnement
conda env export > environment.yml

# Exporter sans les dependances specifiques a l'OS
conda env export --from-history > environment.yml

# Recreer l'environnement
conda env create -f environment.yml

# Lister les environnements
conda env list

# Supprimer un environnement
conda env remove -n ml-project
```

environment.yml

```
name: ml-project
channels:
  - pytorch
  - conda-forge
  - defaults
dependencies:
  - python=3.11
  - numpy=1.26
```

```
  - pandas=2.1
  - scikit-learn=1.3
  - pytorch=2.1
  - pip:
    - mlflow>=2.8      # packages pip si non dispo en conda
    - wandb>=0.16
```

Conda vs pip

- Utilisez Conda pour les dépendances système (CUDA, compilateurs).
- Utilisez pip à l'intérieur d'un environnement Conda pour les packages Python purs.
- Ne mélangez **jamais** `conda install` et `pip install` pour le même package — Conda ne verra pas les packages installés par pip et vice versa.

2.4 Poetry — Gestion moderne des dépendances

Poetry est un gestionnaire de dépendances moderne qui combine la gestion des packages, le verrouillage des versions et la publication de packages dans un seul outil.

Workflow Poetry

```
# Installer Poetry
curl -sSL https://install.python-poetry.org | python3 -

# Creer un nouveau projet
poetry new ml-project
cd ml-project

# Ajouter des dependances
poetry add numpy pandas scikit-learn
poetry add --group dev pytest black mypy

# Installer les dependances
poetry install

# Executer dans l'environnement
poetry run python train.py
poetry run pytest tests/

# Exporter en requirements.txt (compatibilite)
poetry export -f requirements.txt -o requirements.txt
```

pyproject.toml (extrait)

```
[tool.poetry]
name = "ml-project"
version = "0.1.0"
description = "ML classification project"

[tool.poetry.dependencies]
python = "^3.10"
numpy = "^1.26"
pandas = "^2.1"
scikit-learn = "^1.3"

[tool.poetry.group.dev.dependencies]
pytest = "^7.4"
black = "^23.10"
mypy = "^1.6"
```

2.5 Comparaison des outils

Critère	venv+pip	Conda	Poetry	uv
Intégré Python	Oui	Non	Non	Non
Dép. système	Non	Oui	Non	Non
Verrouillage	pip-tools	Oui	Oui	Oui
GPU/CUDA	Manuel	Oui	Manuel	Manuel
Rapidité	Rapide	Lent	Moyen	Très rapide
Publication pkg	Non	Non	Oui	Non
Courbe d'apprentissage	Faible	Moyenne	Moyenne	Faible

2.6 uv — Le nouvel outil rapide

uv est un gestionnaire de packages Python écrit en Rust, compatible avec pip, extrêmement rapide (10–100× plus rapide que pip).

Workflow uv

```
# Installer uv
curl -LsSf https://astral.sh/uv/install.sh | sh

# Creer un environnement et installer
uv venv
source .venv/bin/activate
uv pip install numpy pandas scikit-learn

# Ou tout en un
uv pip install -r requirements.txt
```

```
# Verrouillage
uv pip compile requirements.in -o requirements.txt
uv pip sync requirements.txt
```

2.7 Bonnes pratiques pour la reproductibilité

Règles d'or pour les environnements

1. **Un projet = un environnement** : ne jamais partager d'environnement entre projets.
2. **Fichier de dépendances versionné** : `requirements.txt` ou `environment.yml` dans le dépôt Git.
3. **Épingler les versions** : `numpy==1.26.2`, pas `numpy` sans version.
4. **Séparer dev et prod** : les outils de développement (pytest, black) ne doivent pas être en production.
5. **Documenter la version de Python** : dans le README ou `pyproject.toml`.
6. **Ne jamais modifier l'environnement système** : ne jamais utiliser `sudo pip install`.

Anti-patterns courants

- Installer tous les packages dans l'environnement système
- Utiliser `pip install` sans `-r requirements.txt`
- Oublier de mettre à jour `requirements.txt` après ajout d'un package
- Mélanger Conda et pip de manière anarchique
- Partager un environnement Conda via un chemin absolu

2.8 Mini-projet : environnement reproductible

Mini-projet 2 : Mise en place d'un environnement

En reprenant le projet structuré du chapitre 1 :

1. Créez un environnement virtuel avec `venv` ou Conda.
2. Écrivez un `requirements.in` avec les dépendances directes.
3. Générez un `requirements.txt` verrouillé avec `pip-tools`.
4. Écrivez un `environment.yml` équivalent pour Conda.

5. Vérifiez que votre collègue peut recréer l'environnement à partir du fichier de dépendances uniquement.

6. Ajoutez un script `setup.sh` qui automatise la création de l'environnement.

2.9 Exercices

Exercice 2.1 (★ — Création d'environnement)**.** Créez un environnement virtuel avec chacun des trois outils (venv, Conda, Poetry) pour un projet utilisant `numpy`, `pandas` et `matplotlib`. Comparez les tailles des environnements et les temps de création.

Exercice 2.2 (★★ — Résolution de conflits)**.** Un projet nécessite `tensorflow>=2.15` et `numpy<1.24` (conflit réel). Utilisez `pip-tools` pour identifier le conflit, puis trouvez des versions compatibles. Documentez le processus de résolution.

Exercice 2.3 (★★★ — Environnement multi-plateforme)**.** Créez un projet ML qui fonctionne sur Linux, macOS et Windows :

1. Écrivez un `environment.yml` Conda compatible multi-OS (`conda env export --from-history`)
2. Créez un script `setup.sh` / `setup.bat` qui détecte l'OS et installe les dépendances adaptées
3. Testez sur au moins deux systèmes (ou utilisez des conteneurs Docker pour simuler)
4. Gérez le cas GPU (installer PyTorch CPU vs CUDA selon la machine)

2.10 Aide-mémoire

Résumé du chapitre 2

Outil	Commandes clés
venv	`python -m venv .venv`, `source .venv/bin/activate`
pip-tools	`pip-compile requirements.in`, `pip-sync`
Conda	`conda create -n env python=3.11`, `conda env export`
Poetry	`poetry add pkg`, `poetry install`, `poetry run`
uv	`uv pip install`, `uv pip compile`, `uv pip sync`

Chapitre 3

Contrôle de Version — Git et DVC

3.1 Introduction

Dans tout projet de Machine Learning, la reproductibilité est fondamentale. Sans un système de contrôle de version rigoureux, il devient impossible de retracer quelle version du code, des données et des hyperparamètres a produit un modèle donné. Ce chapitre couvre les outils essentiels : **Git** pour le code et **DVC** (Data Version Control) pour les données et les modèles.

Le piège du « c'était mieux avant »

Sans versionnement, vous ne pourrez jamais revenir à un état antérieur de votre projet. Un modèle performant peut être perdu à jamais si vous écrasez les fichiers sans précaution.

3.2 Git : les fondamentaux pour le ML

3.2.1 Rappels sur Git

Définition 3.1 (Dépôt Git)**.** Un **dépôt Git** (*repository*) est un répertoire dont l'historique complet des modifications est enregistré. Chaque instantané est appelé un **commit**.

Les commandes essentielles :

Commandes Git de base

```
git init                        # Initialiser un d\'ep\^ot
git add <fichier>               # Ajouter au staging
git commit -m "message"         # Cr\'eer un commit
git log --oneline --graph       # Visualiser l'historique
git diff                        # Voir les modifications
```

3.2.2 Branches et fusion

Les branches permettent de travailler sur des fonctionnalités ou des expériences en parallèle sans affecter la branche principale.

Gestion des branches

```
git branch feature/new-model    # Cr\'eer une branche
git checkout feature/new-model  # Basculer sur la branche
git checkout -b feature/xgboost # Cr\'eer et basculer
git merge feature/new-model     # Fusionner dans la branche courante
git branch -d feature/new-model # Supprimer la branche
```

3.2.3 Le fichier `.gitignore` pour le ML

En ML, de nombreux fichiers ne doivent **jamais** être commités :

Exemple de `.gitignore` pour un projet ML

```
# Donn\'ees volumineuses
data/raw/
data/processed/
*.csv
*.parquet
*.h5

# Mod\`eles entra\^in\'es
models/*.pkl
models/*.pt
models/*.onnx

# Environnements
venv/
.conda/
__pycache__/

# Notebooks checkpoints
.ipynb_checkpoints/

# Secrets
.env
*.key
```

3.3 Workflow Git pour les projets ML

Stratégie de branches pour le ML

- `main` : code stable, modèles validés en production.
- `develop` : intégration continue des nouvelles fonctionnalités.
- `feature/*` : nouvelles fonctionnalités ou architectures.
- `experiment/*` : expériences exploratoires (peuvent être abandonnées).

- `data/*` : modifications aux pipelines de données.

Revues de code (Pull Requests) en ML

Une PR en ML devrait inclure :

1. Les métriques avant/après modification.
2. Une justification de l'approche choisie.
3. Les visualisations pertinentes (courbes d'apprentissage, matrices de confusion).
4. La reproductibilité : graines aléatoires fixées, environnement documenté.

3.4 DVC : Data Version Control

3.4.1 Pourquoi Git ne suffit pas

Limites de Git pour les fichiers volumineux

Git stocke l'intégralité de chaque version d'un fichier. Pour un jeu de données de 10 Go modifié 5 fois, le dépôt occuperait 50 Go. Les plateformes comme GitHub imposent une limite de 100 Mo par fichier.

Définition 3.2 (DVC)**. DVC** (Data Version Control) est un outil open-source qui étend Git pour gérer les fichiers volumineux (données, modèles). Il remplace les fichiers par des pointeurs légers (`.dvc`) et stocke le contenu réel sur un stockage distant (S3, GCS, Azure, SSH, etc.).

3.4.2 Installation et initialisation

Installation de DVC

```
pip install dvc[s3]       # Avec support S3
pip install dvc[gs]       # Avec support Google Cloud
pip install dvc[all]      # Tous les backends

# Initialiser DVC dans un d\'ep\^ot Git existant
cd mon-projet-ml
dvc init
git add .dvc .dvcignore
git commit -m "Initialiser DVC"
```

3.4.3 Commandes fondamentales de DVC

Commandes DVC essentielles

Commande	Description
`dvc init`	Initialiser DVC dans un dépôt Git
`dvc add <fichier>`	Suivre un fichier/dossier avec DVC
`dvc push`	Envoyer les données vers le stockage distant
`dvc pull`	Récupérer les données depuis le stockage distant
`dvc remote add`	Configurer un stockage distant
`dvc status`	Vérifier l'état des fichiers suivis
`dvc checkout`	Restaurer les fichiers à partir du cache
`dvc gc`	Nettoyer le cache local

Utilisation de DVC au quotidien

```
# Configurer le stockage distant
dvc remote add -d myremote s3://mon-bucket/dvc-store
git add .dvc/config
git commit -m "Configurer le remote DVC"

# Suivre un jeu de donn\'ees
dvc add data/raw/dataset.csv
git add data/raw/dataset.csv.dvc data/raw/.gitignore
git commit -m "Ajouter dataset v1"

# Envoyer les donn\'ees
dvc push

# R\'ecup\'erer les donn\'ees (apr\`es un clone)
git clone <url-du-repo>
dvc pull
```

3.4.4 Pipelines DVC

DVC permet de définir des pipelines reproductibles via un fichier `dvc.yaml`.

Exemple de `dvc.yaml`

```yaml
stages:
  prepare:
    cmd: python src/prepare.py
    deps:
      - src/prepare.py
      - data/raw/dataset.csv
    outs:
      - data/processed/train.csv
      - data/processed/test.csv
```

```yaml
train:
  cmd: python src/train.py
  deps:
    - src/train.py
    - data/processed/train.csv
  params:
    - train.learning_rate
    - train.n_estimators
  outs:
    - models/model.pkl
  metrics:
    - metrics/scores.json:
        cache: false

evaluate:
  cmd: python src/evaluate.py
  deps:
    - src/evaluate.py
    - models/model.pkl
    - data/processed/test.csv
  metrics:
    - metrics/eval.json:
        cache: false
  plots:
    - metrics/confusion_matrix.csv:
        x: predicted
        y: actual
```

Exécution et comparaison de pipelines

```
# Ex\'ecuter le pipeline complet
dvc repro

# Comparer les m\'etriques entre branches/commits
dvc metrics diff
dvc params diff

# Visualiser le DAG du pipeline
dvc dag
```

3.4.5 Fichier `.dvcignore`

Comme `.gitignore`, le fichier `.dvcignore` indique à DVC les fichiers à ignorer lors du suivi de répertoires :

Exemple de `.dvcignore`

```
# Fichiers temporaires
*.tmp
*.log

# Caches
__pycache__/
.ipynb_checkpoints/

# Fichiers syst\`eme
.DS_Store
Thumbs.db
```

3.5 Architecture Git + DVC

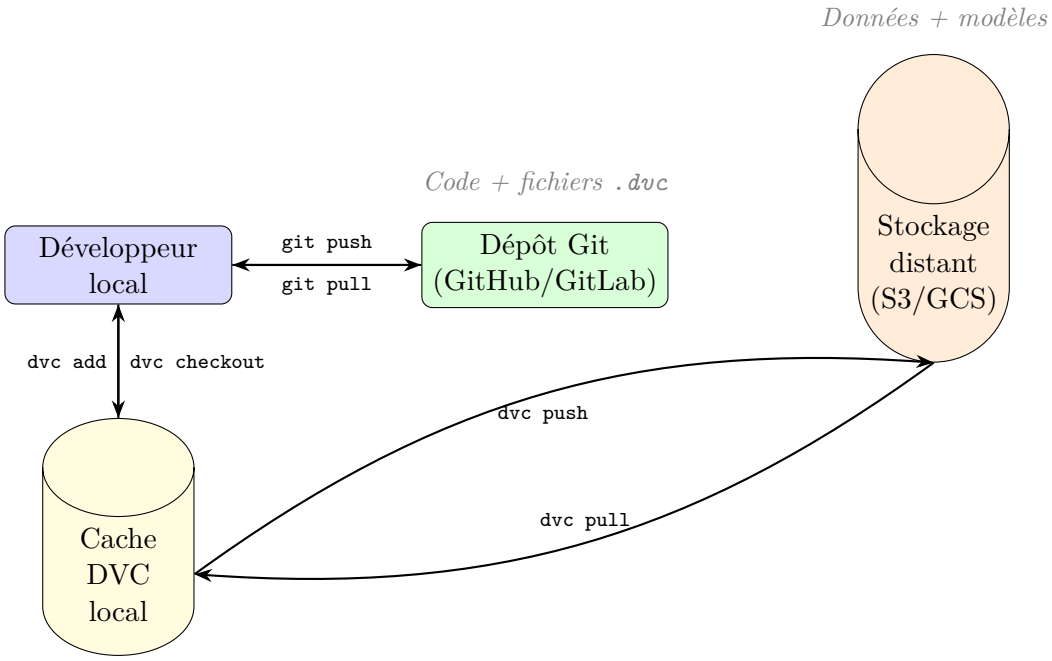


FIG. 3.1 : Architecture du workflow Git + DVC. Le code et les fichiers `.dvc` transitent par Git, tandis que les données volumineuses sont gérées par le cache DVC local et le stockage distant.

3.6 Comparaison des outils de versionnement de données

Remarque 3.3. **Git LFS** est plus simple que DVC mais ne supporte pas les pipelines de données. **LakeFS** offre un véritable branching au niveau des données, ce qui est idéal pour les lacs de données à grande échelle.

TAB. 3.1 : Comparaison des outils de versionnement de données

Critère	DVC	Git LFS	Delta Lake	LakeFS
Licence	Open-source	Open-source	Open-source	Open-source
Stockage	S3, GCS, etc.	Serveur LFS	Object store	S3-compatible
Pipelines	Oui	Non	Non	Non
Branching données	Via Git	Via Git	Non	Oui (natif)
Taille max fichier	Illimitée	Illimitée	Illimitée	Illimitée
Intégration Git	Excellente	Native	Aucune	Bonne
Courbe d'apprentissage	Modérée	Facile	Élevée	Modérée
Cas d'usage idéal	Projets ML	Binaires	Data lakes	Data lakes

3.7 Mini-projet : versionner un dataset et un modèle

Versionner un projet ML complet avec Git et DVC

Objectif : Mettre en place un workflow Git + DVC pour un projet de classification.

1. Initialiser un dépôt Git et DVC :

```
mkdir projet-classification && cd projet-classification
git init && dvc init
dvc remote add -d local_remote /tmp/dvc-storage
```

2. Créer la structure du projet :

```
mkdir -p data/raw data/processed models src metrics
```

3. Ajouter un jeu de données et le suivre avec DVC :

```
# src/download_data.py
from sklearn.datasets import load_wine
import pandas as pd

data = load_wine(as_frame=True)
df = pd.concat([data.data, data.target], axis=1)
df.to_csv("data/raw/wine.csv", index=False)
```

```
python src/download_data.py
dvc add data/raw/wine.csv
git add data/raw/wine.csv.dvc data/raw/.gitignore
git commit -m "Ajouter wine dataset v1"
dvc push
```

4. Créer un pipeline `dvc.yaml` avec les étapes `prepare`, `train` et `evaluate`.

5. Exécuter le pipeline, modifier un hyperparamètre et comparer les résultats :

```
dvc repro
dvc metrics show
# Modifier params.yaml, puis :
dvc repro
dvc metrics diff
```

3.8 Exercices

Exercice 3.1 (Créer un `.gitignore` ML)**. 1/3** Créez un fichier `.gitignore` adapté à un projet de Deep Learning utilisant PyTorch. Incluez les patterns pour les données, les checkpoints, les logs TensorBoard, les environnements virtuels et les fichiers système.

Exercice 3.2 (Pipeline DVC multi-étapes)**. 2/3** Créez un pipeline DVC complet pour un projet de NLP comprenant :

1. Téléchargement des données brutes.
2. Prétraitement (tokenisation, nettoyage).
3. Entraînement d'un modèle de classification de texte.
4. Évaluation avec métriques (accuracy, F1-score).

Utilisez `params.yaml` pour les hyperparamètres et comparez deux configurations avec `dvc metrics diff`.

Exercice 3.3 (Branching et expériences parallèles)**. 3/3** Mettez en place un workflow où deux expériences sont menées en parallèle sur des branches Git différentes. Chaque branche utilise une architecture de modèle différente (par exemple, Random Forest vs. Gradient Boosting). Utilisez DVC pour :

- Suivre les données d'entraînement (partagées entre branches).
- Suivre les modèles entraînés (propres à chaque branche).
- Comparer les métriques entre les deux branches avec `dvc metrics diff`.
- Fusionner la meilleure branche dans `main`.

3.9 Aide-mémoire

Git + DVC — Aide-mémoire

Action	Commande
Initialiser Git + DVC	`git init && dvc init`
Configurer remote DVC	`dvc remote add -d nom s3://bucket/path`
Suivre un fichier	`dvc add data/fichier.csv`
Commiter les pointeurs	`git add fichier.csv.dvc .gitignore`
Envoyer données	`dvc push`
Récupérer données	`dvc pull`
Définir un pipeline	Éditer `dvc.yaml`
Exécuter le pipeline	`dvc repro`
Voir les métriques	`dvc metrics show`
Comparer les métriques	`dvc metrics diff`
Visualiser le DAG	`dvc dag`
Créer une branche	`git checkout -b experiment/nom`
Revue de code	Créer une Pull Request sur GitHub/GitLab

Chapitre 4

Suivi d'Expériences — MLflow, Weights & Biases

4.1 Introduction

Entraîner un modèle de Machine Learning implique de tester des dizaines, voire des centaines, de combinaisons d'hyperparamètres, d'architectures et de jeux de données. Sans un système de suivi structuré, les résultats se perdent et la reproductibilité devient illusoire.

L'anti-pattern du tableur

Beaucoup de data scientists commencent par noter leurs résultats dans un fichier Excel ou Google Sheets :

- Les entrées sont incomplètes (on oublie de noter un paramètre).
- Aucun lien avec le code ou les données utilisés.
- Impossible de reproduire une expérience passée.
- Le fichier devient vite illisible avec des centaines de lignes.

Les outils de suivi d'expériences résolvent tous ces problèmes.

Définition 4.1 (Suivi d'expériences)**.** Le **suivi d'expériences** (*experiment tracking*) consiste à enregistrer automatiquement, pour chaque exécution d'entraînement : les **paramètres**, les **métriques**, les **artefacts** (modèles, graphiques), le **code source** et l'**environnement**.

4.2 MLflow

4.2.1 Présentation

`MLflow` est une plateforme open-source créée par Databricks pour gérer le cycle de vie complet du ML. Elle comprend quatre composants :

Composants de MLflow

MLflow Tracking Enregistrement des paramètres, métriques et artefacts.

MLflow Projects Packaging reproductible du code ML.

MLflow Models Format standard pour le déploiement de modèles.

MLflow Model Registry Gestion du cycle de vie des modèles (staging, production, archivé).

4.2.2 Installation et démarrage

Installation de MLflow

```
pip install mlflow

# Lancer l'interface web
mlflow ui --port 5000
# Accessible sur http://localhost:5000
```

4.2.3 API de suivi (Tracking API)

Concepts fondamentaux de MLflow Tracking

```
import mlflow

# Cr\'eer ou s\'electionner une exp\'erience
mlflow.set_experiment("classification-vin")

# D\'emarrer un run
with mlflow.start_run(run_name="random-forest-v1"):
    # Enregistrer des param\`etres
    mlflow.log_param("n_estimators", 100)
    mlflow.log_param("max_depth", 10)
    mlflow.log_param("random_state", 42)

    # Enregistrer des m\'etriques
    mlflow.log_metric("accuracy", 0.94)
    mlflow.log_metric("f1_score", 0.93)

    # Enregistrer des m\'etriques au fil des \'epoques
    for epoch in range(50):
        mlflow.log_metric("loss", loss_val, step=epoch)

    # Enregistrer des artefacts
    mlflow.log_artifact("plots/confusion_matrix.png")
    mlflow.log_artifact("models/model.pkl")
```

4.2.4 Entraînement complet avec MLflow

Exemple complet : classification avec MLflow

```
import mlflow
import mlflow.sklearn
from sklearn.datasets import load_wine
from sklearn.model_selection import train_test_split
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import accuracy_score, f1_score
import matplotlib.pyplot as plt
from sklearn.metrics import ConfusionMatrixDisplay

# Charger les donn\'ees
X, y = load_wine(return_X_y=True)
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.2, random_state=42
)

# Configuration
params = {
    "n_estimators": 200,
    "max_depth": 15,
    "min_samples_split": 5,
    "random_state": 42,
}

mlflow.set_experiment("wine-classification")

with mlflow.start_run(run_name="rf-optimized"):
    # Enregistrer les param\`etres
    mlflow.log_params(params)

    # Entra\^iner le mod\`ele
    model = RandomForestClassifier(**params)
    model.fit(X_train, y_train)

    # Pr\'edire et \'evaluer
    y_pred = model.predict(X_test)
    acc = accuracy_score(y_test, y_pred)
    f1 = f1_score(y_test, y_pred, average="weighted")

    mlflow.log_metric("accuracy", acc)
    mlflow.log_metric("f1_weighted", f1)

    # Matrice de confusion
    fig, ax = plt.subplots()
    ConfusionMatrixDisplay.from_predictions(
        y_test, y_pred, ax=ax
    )
    fig.savefig("confusion_matrix.png")
```

```
    mlflow.log_artifact("confusion_matrix.png")

    # Enregistrer le mod\`ele
    mlflow.sklearn.log_model(model, "random-forest-model")

    print(f"Accuracy: {acc:.4f}, F1: {f1:.4f}")
```

4.2.5 MLflow Model Registry

Utilisation du Model Registry

```
import mlflow

# Enregistrer un mod\`ele dans le registry
model_uri = "runs:/<run_id>/random-forest-model"
mlflow.register_model(model_uri, "WineClassifier")

# Charger un mod\`ele depuis le registry
from mlflow import MlflowClient

client = MlflowClient()

# Promouvoir en staging
client.transition_model_version_stage(
    name="WineClassifier",
    version=1,
    stage="Staging"
)

# Promouvoir en production
client.transition_model_version_stage(
    name="WineClassifier",
    version=1,
    stage="Production"
)

# Charger le mod\`ele en production
model = mlflow.pyfunc.load_model(
    "models:/WineClassifier/Production"
)
```

Workflow du Model Registry

1. **None** : le modèle vient d'être enregistré.
2. **Staging** : le modèle est en cours de validation.
3. **Production** : le modèle est déployé en production.

4. **Archived** : le modèle est retiré mais conservé.

4.3 Weights & Biases (W&B)

4.3.1 Présentation

`Weights & Biases` (W&B) est une plateforme cloud de suivi d'expériences offrant une interface web riche, des visualisations interactives et des fonctionnalités de collaboration avancées.

Installation et authentification

```
pip install wandb
wandb login  # Saisir la cl\'e API depuis wandb.ai
```

4.3.2 API de suivi W&B

Entraînement complet avec W&B

```python
import wandb
from sklearn.datasets import load_wine
from sklearn.model_selection import train_test_split
from sklearn.ensemble import GradientBoostingClassifier
from sklearn.metrics import accuracy_score, f1_score

# Initialiser W&B
wandb.init(
    project="wine-classification",
    name="gradient-boosting-v1",
    config={
        "n_estimators": 150,
        "learning_rate": 0.1,
        "max_depth": 5,
        "random_state": 42,
    }
)
config = wandb.config

# Charger les donn\'ees
X, y = load_wine(return_X_y=True)
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.2, random_state=config.random_state
)

# Entra\^iner le mod\`ele
model = GradientBoostingClassifier(
    n_estimators=config.n_estimators,
    learning_rate=config.learning_rate,
```

```
    max_depth=config.max_depth,
    random_state=config.random_state,
)

# Suivi \'epoque par \'epoque (staged_predict)
model.fit(X_train, y_train)
for i, y_pred_staged in enumerate(
    model.staged_predict(X_test)
):
    acc = accuracy_score(y_test, y_pred_staged)
    wandb.log({"epoch": i, "accuracy": acc})

# M\'etriques finales
y_pred = model.predict(X_test)
final_acc = accuracy_score(y_test, y_pred)
final_f1 = f1_score(y_test, y_pred, average="weighted")

wandb.log({
    "final_accuracy": final_acc,
    "final_f1": final_f1,
})

# Enregistrer le mod\`ele comme artefact
artifact = wandb.Artifact("wine-model", type="model")
artifact.add_file("models/model.pkl")
wandb.log_artifact(artifact)

wandb.finish()
```

4.3.3 Sweeps : recherche d'hyperparamètres

Configuration d'un Sweep W&B

```yaml
# sweep_config.yaml
program: train.py
method: bayes  # random, grid, bayes
metric:
  name: final_accuracy
  goal: maximize
parameters:
  n_estimators:
    values: [50, 100, 200, 500]
  learning_rate:
    min: 0.001
    max: 0.3
  max_depth:
    values: [3, 5, 7, 10]
```

Lancer un Sweep

```
# Cr\'eer le sweep
wandb sweep sweep_config.yaml
# Retourne un sweep_id

# Lancer les agents
wandb agent <username>/<project>/<sweep_id>
```

4.4 Hydra pour la gestion de configuration

Hydra est un framework de configuration développé par Meta qui simplifie la gestion des paramètres dans les projets ML.

Structure de configuration Hydra

```yaml
# config/config.yaml
defaults:
  - model: random_forest
  - data: wine

training:
  test_size: 0.2
  random_state: 42

# config/model/random_forest.yaml
name: RandomForest
n_estimators: 100
max_depth: 10

# config/model/gradient_boosting.yaml
name: GradientBoosting
n_estimators: 200
learning_rate: 0.1

# config/data/wine.yaml
name: wine
path: data/raw/wine.csv
```

Utilisation de Hydra avec MLflow

```python
import hydra
from omegaconf import DictConfig
import mlflow

@hydra.main(config_path="config", config_name="config",
            version_base=None)
def train(cfg: DictConfig):
```

```
    mlflow.set_experiment(cfg.data.name)

    with mlflow.start_run():
        # Les param\`etres viennent de la config Hydra
        mlflow.log_params({
            "model": cfg.model.name,
            "n_estimators": cfg.model.n_estimators,
            "test_size": cfg.training.test_size,
        })

        # ... entra\^inement ...

        mlflow.log_metric("accuracy", accuracy)

if __name__ == "__main__":
    train()
```

Override de configuration en ligne de commande

```
# Utiliser la config par d\'efaut
python train.py

# Changer de mod\`ele
python train.py model=gradient_boosting

# Override un param\`etre
python train.py model.n_estimators=500

# Multi-run (grid search)
python train.py --multirun \
    model=random_forest,gradient_boosting \
    model.n_estimators=100,200,500
```

4.5 Architecture du suivi d'expériences

4.6 Comparaison des outils de suivi

Remarque 4.2. **MLflow** est le choix privilégié pour les équipes souhaitant garder le contrôle total de leur infrastructure (self-hosted). **W&B** offre la meilleure expérience utilisateur pour les équipes de recherche qui préfèrent une solution cloud clé en main.

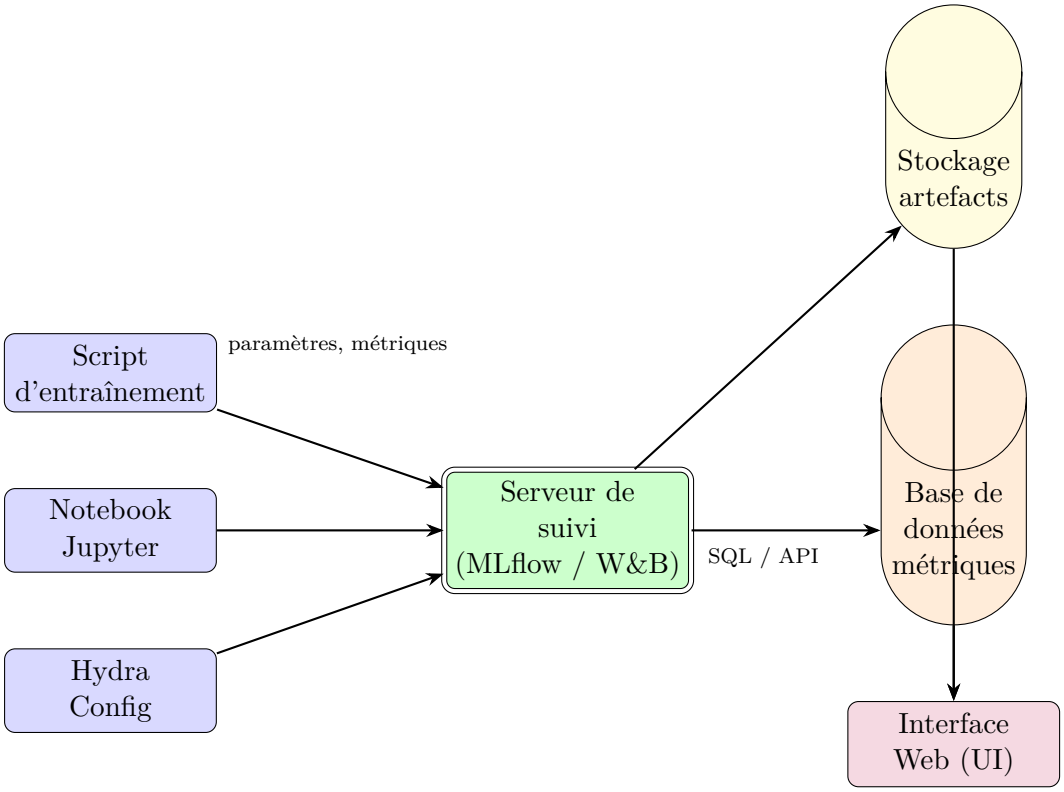


FIG. 4.1 : Architecture générale d'un système de suivi d'expériences. Les scripts et notebooks envoient les données au serveur de suivi, qui les persiste dans une base de données et un stockage d'artefacts, accessibles via une interface web.

TAB. 4.1 : Comparaison des outils de suivi d'expériences

Critère	**MLflow**	**W&B**	**Neptune**	**ClearML**
Licence	Open-source	Freemium	Freemium	Open-source
Hébergement	Self-hosted	Cloud	Cloud	Self/Cloud
Interface web	Bonne	Excellente	Excellente	Bonne
Model Registry	Oui	Oui	Oui	Oui
Sweeps/HPO	Non natif	Oui	Oui	Oui
Collaboration	Basique	Avancée	Avancée	Bonne
Intégration DL	Bonne	Excellente	Excellente	Bonne
Taille équipe	Toutes	Petite–grande	Moyenne–grande	Toutes
Coût production	Gratuit	Payant	Payant	Gratuit

4.7 Mini-projet : suivi d'expériences pour le projet fil rouge

Instrumenter le projet des chapitres 1–3 avec MLflow

Objectif : Ajouter le suivi d'expériences au pipeline de classification créé dans les chapitres précédents.

1. Installer et lancer MLflow :

```
pip install mlflow
mlflow ui --port 5000 &
```

2. Modifier le script d'entraînement pour intégrer MLflow :

```
import mlflow
import mlflow.sklearn
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import accuracy_score, f1_score
import json

mlflow.set_experiment("projet-fil-rouge")

with mlflow.start_run(run_name="baseline-rf"):
    params = {
        "n_estimators": 100,
        "max_depth": 10,
        "random_state": 42,
    }
    mlflow.log_params(params)

    model = RandomForestClassifier(**params)
    model.fit(X_train, y_train)

    y_pred = model.predict(X_test)
    metrics = {
        "accuracy": accuracy_score(y_test, y_pred),
        "f1": f1_score(y_test, y_pred, average="weighted"),
    }
    mlflow.log_metrics(metrics)

    # Sauvegarder les m\'etriques pour DVC
    with open("metrics/scores.json", "w") as f:
        json.dump(metrics, f, indent=2)

    mlflow.sklearn.log_model(model, "model")
```

3. Créer une configuration Hydra pour varier les hyperparamètres.

4. Lancer au moins 5 expériences avec des paramètres différents.
5. Utiliser l'interface MLflow pour comparer les runs et identifier la meilleure configuration.
6. Enregistrer le meilleur modèle dans le Model Registry et le promouvoir en `Staging`.

4.8 Exercices

Exercice 4.1 (Prise en main de MLflow)**. 1/3** Installez MLflow et enregistrez une expérience simple :

1. Entraînez un modèle `LogisticRegression` sur le dataset Iris.
2. Enregistrez les paramètres `C`, `solver` et `max_iter`.
3. Enregistrez les métriques `accuracy` et `f1_macro`.
4. Lancez l'interface MLflow et vérifiez que le run apparaît.
5. Relancez avec des paramètres différents et comparez dans l'UI.

Exercice 4.2 (Sweeps W&B et analyse)**. 2/3** Utilisez W&B Sweeps pour optimiser les hyperparamètres d'un modèle `GradientBoosting` :

1. Créez un fichier `sweep_config.yaml` explorant :
   - `n_estimators` : 50, 100, 200, 500.
   - `learning_rate` : entre 0.001 et 0.5 (échelle log).
   - `max_depth` : 3, 5, 7, 10.
2. Lancez un sweep bayésien avec 20 runs.
3. Analysez les résultats dans l'interface W&B : parallel coordinates, importance des hyperparamètres.
4. Rédigez un rapport W&B résumant vos conclusions.

Exercice 4.3 (Pipeline complet Hydra + MLflow + DVC)**. 3/3** Construisez un pipeline ML complet intégrant les trois outils :

1. Utilisez **DVC** pour versionner les données et définir les étapes du pipeline (`dvc.yaml`).
2. Utilisez **Hydra** pour gérer les configurations : créez au moins trois fichiers de config (un par modèle).
3. Utilisez **MLflow** pour suivre chaque expérience automatiquement.
4. Intégrez le tout pour que `dvc repro` exécute le pipeline et enregistre les résultats dans MLflow.
5. Comparez les résultats de trois architectures différentes et enregistrez la meilleure dans le Model Registry.

4.9 Aide-mémoire

Suivi d'expériences — Aide-mémoire

Action	Commande / Code
MLflow	
Lancer l'UI	`mlflow ui --port 5000`
Créer une expérience	`mlflow.set_experiment("nom")`
Démarrer un run	`mlflow.start_run(run_name="nom")`
Logger paramètres	`mlflow.log_params(dict)`
Logger métriques	`mlflow.log_metric("nom", val, step)`
Logger un modèle	`mlflow.sklearn.log_model(m, "nom")`
Enregistrer au registry	`mlflow.register_model(uri, "nom")`
Weights & Biases	
Connexion	`wandb login`
Initialiser	`wandb.init(project, name, config)`
Logger métriques	`wandb.log({"metric": val})`
Logger artefact	`wandb.log_artifact(artifact)`
Créer un sweep	`wandb sweep config.yaml`
Lancer un agent	`wandb agent user/proj/sweep_id`
Terminer	`wandb.finish()`
Hydra	
Décorateur principal	`@hydra.main(config_path, config_name)`
Override paramètre	`python train.py model.lr=0.01`
Multi-run	`python train.py --multirun model=a,b,c`

Chapitre 5

Pipelines de Données et Feature Stores

5.1 Qu'est-ce qu'un pipeline de données ?

Un **pipeline de données** est une séquence automatisée d'étapes qui transforme des données brutes en données exploitables par un modèle de machine learning. Dans un contexte MLOps, les pipelines assurent la *reproductibilité*, la *traçabilité* et l'*automatisation* du flux de données.

Définition 5.1 (Pipeline de données)**.** Un pipeline de données est un ensemble d'étapes de traitement orchestrées, où chaque étape consomme des entrées, effectue une transformation, et produit des sorties. Les étapes sont **idémpotentes** (exécuter deux fois donne le même résultat) et **déterministes** (mêmes entrées ⇒ mêmes sorties).

5.1.1 ETL vs ELT

Deux paradigmes classiques structurent les pipelines de données :

	ETL	**ELT**
Principe	Extract → Transform → Load	Extract → Load → Transform
Transformation	Avant le chargement (en mémoire)	Après le chargement (dans le data warehouse)
Outils	Airflow, Prefect, Luigi	dbt, Spark SQL, BigQuery
Cas d'usage	Données structurées, volume modéré	Grands volumes, stockage bon marché

Remarque 5.2. En MLOps, on utilise souvent un hybride : ELT pour la préparation générale des données, puis un pipeline ML spécifique (feature engineering, entraînement, évaluation) orchestré séparément.

5.2 Outils d'orchestration

5.2.1 Apache Airflow

Airflow est l'orchestrateur le plus répandu. Il utilise des **DAGs** (graphes orientés acycliques) définis en Python pour décrire les dépendances entre tâches.

Exemple de DAG Airflow

```python
from airflow import DAG
from airflow.operators.python import PythonOperator
from datetime import datetime

def extract():
    """Extract data from source."""
    print("Extracting data...")

def transform():
    """Transform raw data."""
    print("Transforming data...")

def load():
    """Load data to destination."""
    print("Loading data...")

with DAG(
    dag_id="ml_data_pipeline",
    start_date=datetime(2025, 1, 1),
    schedule_interval="@daily",
    catchup=False,
) as dag:
    t1 = PythonOperator(task_id="extract", python_callable=extract)
    t2 = PythonOperator(task_id="transform", python_callable=transform)
    t3 = PythonOperator(task_id="load", python_callable=load)

    t1 >> t2 >> t3  # dependances
```

5.2.2 Prefect

Prefect adopte une approche plus pythonique avec des décorateurs `@flow` et `@task`, sans la complexité de configuration d'Airflow.

5.2.3 Luigi

Luigi (par Spotify) est orienté cibles : chaque tâche définit une sortie (`output()`) et ses dépendances (`requires()`). Le graphe est construit automatiquement.

5.3 Écrire un pipeline avec Prefect

Pipeline ML complet avec Prefect

```python
from prefect import flow, task
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.ensemble import RandomForestClassifier
```

```python
from sklearn.metrics import accuracy_score, f1_score
import joblib
from pathlib import Path


@task(retries=2, retry_delay_seconds=10)
def extract_data(path: str) -> pd.DataFrame:
    """Load raw data from CSV."""
    df = pd.read_csv(path)
    print(f"Loaded {len(df)} rows from {path}")
    return df


@task
def clean_data(df: pd.DataFrame) -> pd.DataFrame:
    """Remove duplicates and handle missing values."""
    df = df.drop_duplicates()
    df = df.dropna(subset=["target"])
    df = df.fillna(df.median(numeric_only=True))
    print(f"After cleaning: {len(df)} rows")
    return df


@task
def feature_engineering(df: pd.DataFrame) -> pd.DataFrame:
    """Create new features."""
    numeric_cols = df.select_dtypes(include="number").columns
    for col in numeric_cols:
        if col != "target":
            df[f"{col}_squared"] = df[col] ** 2
            df[f"{col}_log"] = df[col].clip(lower=1e-6).apply(
                lambda x: x ** 0.5
            )
    return df


@task
def split(df: pd.DataFrame, test_size: float = 0.2):
    """Split data into train and test sets."""
    X = df.drop("target", axis=1)
    y = df["target"]
    return train_test_split(X, y, test_size=test_size, random_state=42)


@task
def train_model(X_train, y_train, n_estimators: int = 100):
    """Train a RandomForest classifier."""
    model = RandomForestClassifier(
        n_estimators=n_estimators, random_state=42, n_jobs=-1
    )
    model.fit(X_train, y_train)
```

```
    return model


@task
def evaluate_model(model, X_test, y_test) -> dict:
    """Evaluate model and return metrics."""
    y_pred = model.predict(X_test)
    metrics = {
        "accuracy": accuracy_score(y_test, y_pred),
        "f1_score": f1_score(y_test, y_pred, average="weighted"),
    }
    print(f"Metrics: {metrics}")
    return metrics


@task
def save_model(model, path: str) -> None:
    """Save trained model to disk."""
    Path(path).parent.mkdir(parents=True, exist_ok=True)
    joblib.dump(model, path)
    print(f"Model saved to {path}")


@flow(name="ml-training-pipeline")
def training_pipeline(
    data_path: str = "data/dataset.csv",
    model_path: str = "models/model.joblib",
    n_estimators: int = 100,
):
    """End-to-end ML training pipeline."""
    # Extract
    raw_df = extract_data(data_path)

    # Transform
    clean_df = clean_data(raw_df)
    feat_df = feature_engineering(clean_df)

    # Split and train
    X_train, X_test, y_train, y_test = split(feat_df)
    model = train_model(X_train, y_train, n_estimators)

    # Evaluate and save
    metrics = evaluate_model(model, X_test, y_test)
    save_model(model, model_path)

    return metrics


if __name__ == "__main__":
    training_pipeline()
```

Exécution et déploiement Prefect

```
# Installer Prefect
pip install prefect

# Executer localement
python pipeline.py

# Demarrer le serveur Prefect (UI)
prefect server start

# Deployer le flow
prefect deploy pipeline.py:training_pipeline \
    --name "ml-training" \
    --cron "0 6 * * *"  # tous les jours a 6h
```

5.4 DVC Pipelines : orchestrer un pipeline ML

DVC (Data Version Control) permet de définir des pipelines ML dans un fichier `dvc.yaml`. Chaque étape spécifie ses dépendances et ses sorties, et DVC ne ré-exécute que les étapes dont les entrées ont changé.

dvc.yaml — Pipeline ML

```
stages:
  prepare:
    cmd: python src/prepare.py --input data/raw.csv
         --output data/prepared.csv
    deps:
      - src/prepare.py
      - data/raw.csv
    outs:
      - data/prepared.csv

  featurize:
    cmd: python src/featurize.py --input data/prepared.csv
         --output data/features.csv
    deps:
      - src/featurize.py
      - data/prepared.csv
    outs:
      - data/features.csv

  train:
    cmd: python src/train.py --input data/features.csv
         --model models/model.pkl
    deps:
      - src/train.py
      - data/features.csv
```

```
    outs:
      - models/model.pkl
    params:
      - train.n_estimators
      - train.max_depth

  evaluate:
    cmd: python src/evaluate.py --model models/model.pkl
         --data data/features.csv
    deps:
      - src/evaluate.py
      - models/model.pkl
      - data/features.csv
    metrics:
      - metrics.json:
          cache: false
    plots:
      - plots/confusion_matrix.png
```

Commandes DVC pipelines

```
# Executer le pipeline complet
dvc repro

# Visualiser le graphe du pipeline
dvc dag

# Voir les metriques
dvc metrics show

# Comparer avec une autre branche
dvc metrics diff main
```

Quand utiliser DVC vs Prefect/Airflow

- **DVC** : pipelines ML liés au code (même dépôt Git), exécution locale ou CI/CD, versionnement des données intégré.
- **Prefect/Airflow** : orchestration de flux complexes, exécution planifiée, dépendances entre systèmes (API, bases de données), monitoring avancé.

5.5 Feature Stores

5.5.1 Pourquoi un feature store ?

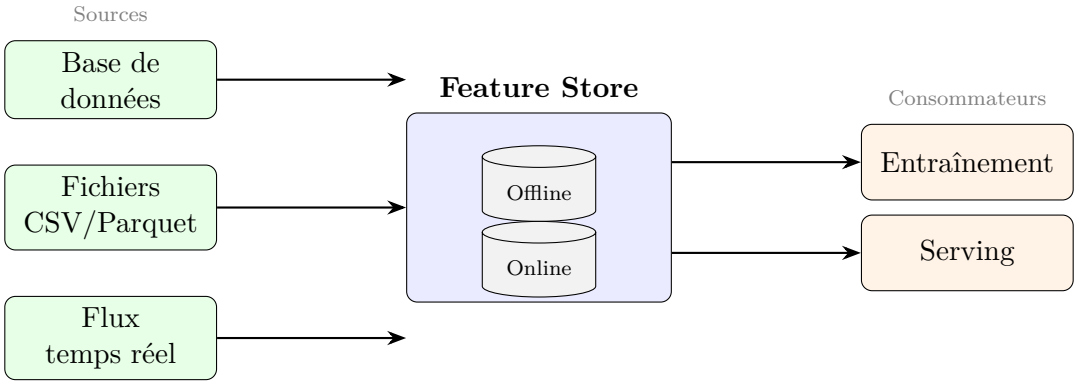
Le **feature engineering** est souvent la partie la plus coûteuse d'un projet ML. Sans feature store :

- Les features sont recalculées à chaque expérience

- Le code de feature engineering est dupliqué entre équipes
- Les features d'entraînement et de serving diffèrent (*training-serving skew*)
- Pas de traçabilité sur l'origine des features

Définition 5.3 (Feature Store)**.** Un **feature store** est une plateforme centralisée pour stocker, gérer, découvrir et servir des features ML. Il garantit la **cohérence** entre l'entraînement (accès *batch*) et l'inférence (accès *temps réel*).

5.5.2 Architecture d'un feature store



5.5.3 Feast : un feature store open-source

Feast est le feature store open-source le plus populaire. Il gère les définitions de features, le stockage offline (fichiers Parquet, BigQuery) et online (Redis, DynamoDB).

Définition de features avec Feast

```python
# feature_repo/features.py
from feast import Entity, FeatureView, Field, FileSource
from feast.types import Float32, Int64
from datetime import timedelta

# Source de donnees
customer_source = FileSource(
    path="data/customer_features.parquet",
    timestamp_field="event_timestamp",
)

# Entite
customer = Entity(
    name="customer_id",
    join_keys=["customer_id"],
)

# Vue de features
customer_features = FeatureView(
    name="customer_features",
    entities=[customer],
```

```
    ttl=timedelta(days=1),
    schema=[
        Field(name="total_purchases", dtype=Int64),
        Field(name="avg_order_value", dtype=Float32),
        Field(name="days_since_last_order", dtype=Int64),
        Field(name="purchase_frequency", dtype=Float32),
    ],
    source=customer_source,
    online=True,
)
```

Utilisation de Feast

```
from feast import FeatureStore
import pandas as pd

store = FeatureStore(repo_path="feature_repo/")

# Recuperer des features pour l'entrainement (offline)
entity_df = pd.DataFrame({
    "customer_id": [1001, 1002, 1003],
    "event_timestamp": pd.to_datetime(["2025-01-15"] * 3),
})
training_df = store.get_historical_features(
    entity_df=entity_df,
    features=[
        "customer_features:total_purchases",
        "customer_features:avg_order_value",
        "customer_features:purchase_frequency",
    ],
).to_df()

# Recuperer des features en temps reel (online)
online_features = store.get_online_features(
    features=[
        "customer_features:total_purchases",
        "customer_features:avg_order_value",
    ],
    entity_rows=[{"customer_id": 1001}],
).to_dict()
```

Commandes Feast

```
# Installer Feast
pip install feast

# Initialiser un projet
feast init feature_repo
cd feature_repo
```

```
# Appliquer les definitions
feast apply

# Materialiser les features dans le store online
feast materialize 2025-01-01T00:00:00 2025-12-31T23:59:59

# Lister les feature views
feast feature-views list
```

5.6 Bonnes pratiques de feature engineering

Règles d'or du feature engineering

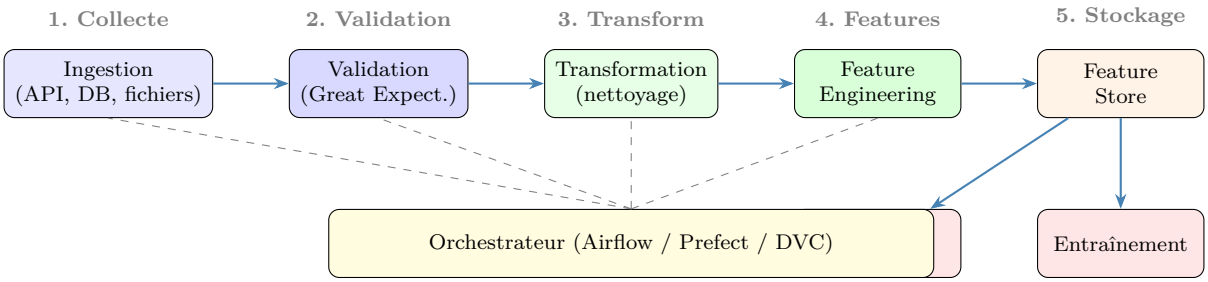
1. **Centraliser les définitions** : une seule source de vérité pour chaque feature (feature store ou module Python dédié).
2. **Versionner les transformations** : le code de feature engineering est versionné dans Git comme tout autre code.
3. **Tester les features** : vérifier les distributions, les valeurs manquantes, les bornes attendues.
4. **Documenter chaque feature** : nom, description, unité, source, fréquence de mise à jour.
5. **Éviter le data leakage** : ne jamais utiliser d'information future lors du calcul des features.
6. **Garantir la cohérence train/serve** : utiliser le **même code** pour calculer les features en entraînement et en inférence.

Anti-patterns du feature engineering

- Calculer des features directement dans le notebook sans les centraliser
- Utiliser des transformations différentes en entraînement et en production (training-serving skew)
- Ne pas surveiller la distribution des features en production (feature drift)
- Créer des centaines de features sans évaluer leur importance
- Hardcoder des seuils de discrétisation sans les paramétrer

5.7 Architecture d'un pipeline de données ML

Le diagramme suivant montre l'architecture complète d'un pipeline de données dans un système MLOps :



5.8 Comparaison des outils d'orchestration

Critère	Airflow	Prefect	Dagster	Luigi
Facilité d'installation	Difficile	Facile	Moyen	Facile
Courbe d'apprentissage	Raide	Douce	Moyenne	Douce
UI intégrée	Oui	Oui	Oui	Basique
Gestion des erreurs	Basique	Avancée	Avancée	Basique
Scalabilité	Très haute	Haute	Haute	Moyenne
Cloud natif	Non	Oui	Non	Non
Communauté	Très large	Croissante	Croissante	Stable
Cas d'usage ML	Généraliste	ML-friendly	ML-friendly	ETL
Typage des données	Non	Oui	Oui	Non

Choisir le bon outil

- **Airflow** : choix éprouvé pour les grandes organisations avec une équipe d'infrastructure dédiée.
- **Prefect** : idéal pour les équipes ML qui veulent démarrer rapidement sans infrastructure lourde.
- **Dagster** : excellent pour les pipelines de données typés avec des assets matérialisés.
- **Luigi** : adapté aux pipelines simples et linéaires.

5.9 Mini-projet : pipeline de données pour le projet de cours

Mini-projet 5 : Pipeline de données complet

Construisez un pipeline de données pour le projet de cours :

1. **Ingestion** : écrivez une tâche Prefect qui télécharge un jeu de données depuis une URL (par exemple, un dataset UCI ou Kaggle).
2. **Validation** : ajoutez une étape qui vérifie le schéma des données (colonnes attendues, types, pas de valeurs aberrantes).
3. **Transformation** : nettoyez les données (valeurs manquantes, doublons, en-

codage des variables catégorielles).

4. **Feature engineering** : créez au moins 5 features dérivées pertinentes pour votre problème.
5. **DVC pipeline** : écrivez un `dvc.yaml` qui reproduit le même pipeline avec DVC.
6. **Bonus** : configurez Feast pour stocker et servir vos features.

Livrables : code source, `dvc.yaml`, documentation des features.

5.10 Exercices

Exercice 5.1 (★ — Pipeline Prefect simple)**.** Écrivez un pipeline Prefect à trois étapes (extract, transform, load) qui :

1. Charge un fichier CSV
2. Supprime les lignes avec des valeurs manquantes
3. Sauvegarde le résultat en Parquet

Testez le pipeline localement et vérifiez les logs dans le terminal.

Exercice 5.2 (★★ — Pipeline DVC multi-étapes)**.** Créez un pipeline DVC complet pour un problème de classification :

1. Définissez 4 étapes dans `dvc.yaml` : prepare, featurize, train, evaluate
2. Utilisez `params.yaml` pour les hyperparamètres
3. Exécutez `dvc repro` et vérifiez que seules les étapes modifiées sont ré-exécutées
4. Comparez les métriques entre deux configurations avec `dvc metrics diff`

Exercice 5.3 (★★★ — Feature store avec Feast)**.** Mettez en place un feature store complet :

1. Créez un projet Feast avec au moins deux `FeatureView`
2. Définissez des features pour deux entités différentes (par exemple, utilisateur et produit)
3. Matérialisez les features dans le store online
4. Écrivez un script d'entraînement qui récupère les features historiques via `get_historical_featur`
5. Écrivez un script de serving qui récupère les features en temps réel via `get_online_features`
6. Vérifiez que les features sont identiques entre entraînement et serving (pas de training-serving skew)

5.11 Aide-mémoire

Résumé du chapitre 5

Concept	Point clé
Pipeline de données	Séquence automatisée : ingestion → validation → transformation → features
ETL vs ELT	ETL : transformer avant le chargement ; ELT : transformer après
Prefect	`@flow`, `@task`, retries, UI intégrée
DVC pipelines	`dvc.yaml`, `dvc repro`, ne ré-exécute que le nécessaire
Feature store	Centralise les features, garantit la cohérence train/serve
Feast	`feast apply`, `get_historical_features`, `get_online_features`
Training-serving skew	Différence entre features en entraînement et en production

Chapitre 6

Entraînement à Grande Échelle — Distributed Training

6.1 Pourquoi l'entraînement distribué ?

Les modèles modernes de deep learning atteignent des tailles qui rendent l'entraînement sur un seul GPU impossible ou prohibitivement lent :

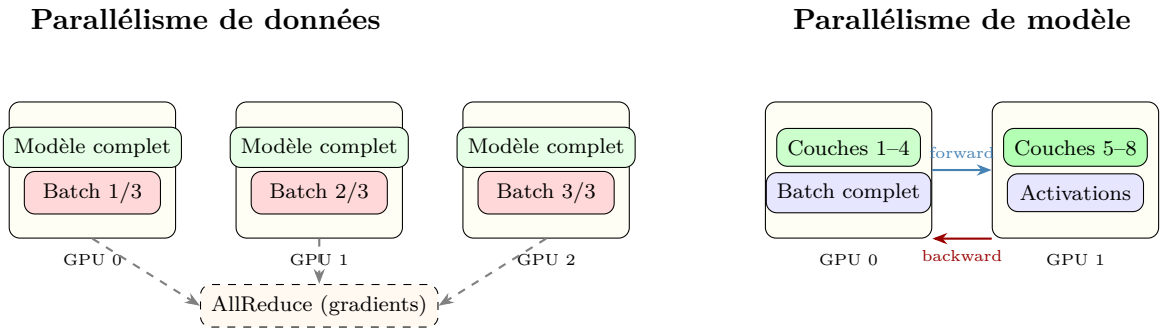
- **GPT-3** : 175 milliards de paramètres
- **LLaMA-2 70B** : 70 milliards de paramètres
- **Vision Transformers** : entraînement sur des millions d'images en haute résolution

L'entraînement distribué répartit le travail sur plusieurs GPU (ou plusieurs machines) pour réduire le temps d'entraînement et permettre l'entraînement de modèles qui ne tiennent pas en mémoire sur un seul accélérateur.

6.1.1 Parallélisme de données vs parallélisme de modèle

Définition 6.1 (Parallélisme de données)**.** Chaque GPU possède une **copie complète** du modèle. Le batch de données est divisé en sous-batches, chaque GPU traite un sous-batch, puis les gradients sont agrégés (moyenne) avant la mise à jour des poids.

Définition 6.2 (Parallélisme de modèle)**.** Le modèle est **partitionné** entre plusieurs GPU. Chaque GPU ne contient qu'une partie du modèle (par couches ou par tenseurs). Les activations sont transmises entre GPU lors du forward pass.



6.2 Formulation mathématique du SGD parallèle

Soit N le nombre de GPU (workers). À chaque itération :

SGD avec parallélisme de données

1. Le batch global $\mathcal{B}$ de taille B est divisé en N sous-batches $\mathcal{B}_k$ de taille B/N.
2. Chaque worker k calcule son gradient local :

$$g_k = \frac{1}{|\mathcal{B}_k|} \sum_{(x,y)\in\mathcal{B}_k} \nabla_\theta \, \ell(f_\theta(x), y)$$

3. Les gradients sont agrégés par AllReduce :

$$\bar{g} = \frac{1}{N} \sum_{k=1}^{N} g_k$$

4. Chaque worker met à jour ses poids :

$$\theta \leftarrow \theta - \eta \, \bar{g}$$

Le gradient agrégé $\bar{g}$ est un estimateur non biaisé du gradient sur le batch complet. Le batch effectif est de taille $B = N \cdot B_{\text{local}}$.

Scaling du learning rate

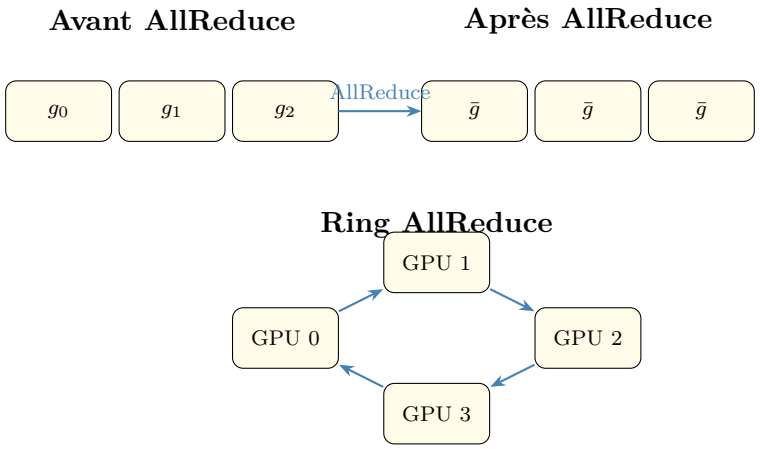
Avec N GPU, le batch effectif est multiplié par N. La règle empirique (*linear scaling rule*) :

$$\eta_{\text{distribué}} = N \cdot \eta_{\text{base}}$$

En pratique, on utilise un **warmup linéaire** sur les premières itérations pour stabiliser l'entraînement avec un grand learning rate.

6.2.1 AllReduce

L'opération **AllReduce** est le cœur de la synchronisation des gradients. Elle agrège les tenseurs de tous les workers et distribue le résultat à chacun.



6.3 PyTorch DistributedDataParallel (DDP)

DDP est le mécanisme standard de PyTorch pour le parallélisme de données. Chaque processus gère un GPU, et les gradients sont synchronisés automatiquement via NCCL.

Entraînement DDP complet

```python
"""Distributed Data Parallel training with PyTorch."""
import os
import torch
import torch.nn as nn
import torch.optim as optim
import torch.distributed as dist
from torch.nn.parallel import DistributedDataParallel as DDP
from torch.utils.data import DataLoader, DistributedSampler
from torchvision import datasets, transforms


def setup(rank: int, world_size: int) -> None:
    """Initialize the distributed process group."""
    os.environ["MASTER_ADDR"] = "localhost"
    os.environ["MASTER_PORT"] = "12355"
    dist.init_process_group("nccl", rank=rank, world_size=world_size)
    torch.cuda.set_device(rank)


def cleanup() -> None:
    """Destroy the process group."""
    dist.destroy_process_group()


class SimpleCNN(nn.Module):
    def __init__(self, num_classes: int = 10):
        super().__init__()
        self.features = nn.Sequential(
            nn.Conv2d(1, 32, 3, padding=1), nn.ReLU(),
            nn.MaxPool2d(2),
```

```
            nn.Conv2d(32, 64, 3, padding=1), nn.ReLU(),
            nn.MaxPool2d(2),
        )
        self.classifier = nn.Sequential(
            nn.Flatten(),
            nn.Linear(64 * 7 * 7, 128), nn.ReLU(),
            nn.Linear(128, num_classes),
        )

    def forward(self, x):
        return self.classifier(self.features(x))


def train(rank: int, world_size: int, epochs: int = 10) -> None:
    """Training loop for one process (one GPU)."""
    setup(rank, world_size)

    # Data
    transform = transforms.Compose([
        transforms.ToTensor(),
        transforms.Normalize((0.1307,), (0.3081,)),
    ])
    dataset = datasets.MNIST(
        "data", train=True, download=True, transform=transform
    )
    sampler = DistributedSampler(
        dataset, num_replicas=world_size, rank=rank, shuffle=True
    )
    loader = DataLoader(
        dataset, batch_size=64, sampler=sampler, num_workers=2,
        pin_memory=True,
    )

    # Model
    model = SimpleCNN().to(rank)
    model = DDP(model, device_ids=[rank])

    # Optimizer
    optimizer = optim.Adam(model.parameters(), lr=1e-3)
    criterion = nn.CrossEntropyLoss()

    # Training loop
    for epoch in range(epochs):
        sampler.set_epoch(epoch)  # important for shuffling
        model.train()
        total_loss = 0.0

        for batch_idx, (data, target) in enumerate(loader):
            data, target = data.to(rank), target.to(rank)

            optimizer.zero_grad()
```

```
            output = model(data)
            loss = criterion(output, target)
            loss.backward()
            optimizer.step()

            total_loss += loss.item()

        if rank == 0:
            avg_loss = total_loss / len(loader)
            print(f"Epoch {epoch+1}/{epochs}, Loss: {avg_loss:.4f}")

    # Save model (only on rank 0)
    if rank == 0:
        torch.save(model.module.state_dict(), "model_ddp.pt")

    cleanup()


if __name__ == "__main__":
    world_size = torch.cuda.device_count()
    torch.multiprocessing.spawn(
        train, args=(world_size,), nprocs=world_size, join=True
    )
```

Lancement avec torchrun

```
# Sur une seule machine avec 4 GPU
torchrun --nproc_per_node=4 train_ddp.py

# Sur 2 machines avec 4 GPU chacune
# Machine 0 (master)
torchrun --nproc_per_node=4 --nnodes=2 --node_rank=0 \
    --master_addr=192.168.1.1 --master_port=12355 train_ddp.py

# Machine 1
torchrun --nproc_per_node=4 --nnodes=2 --node_rank=1 \
    --master_addr=192.168.1.1 --master_port=12355 train_ddp.py
```

Bonnes pratiques DDP

1. **Appeler `sampler.set_epoch(epoch)`** : sinon le shuffling sera identique à chaque époque.
2. **Sauvegarder via `model.module`** : DDP wrappe le modèle, il faut accéder au module interne.
3. **Sauvegarder uniquement sur rank 0** : évite les écritures concurrentes.
4. **Utiliser `pin_memory=True`** : accélère les transferts CPU → GPU.

5. **Préférer `torchrun`** : gère automatiquement les variables d'environnement (`RANK`, `WORLD_SIZE`).

6.4 Multi-GPU avec Accelerate (Hugging Face)

Accelerate de Hugging Face abstrait la complexité de DDP derrière une API minimale. Le même code fonctionne sur CPU, un GPU, multi-GPU et multi-machines.

Entraînement avec Accelerate

```python
"""Training with Hugging Face Accelerate."""
from accelerate import Accelerator
import torch
import torch.nn as nn
from torch.utils.data import DataLoader
from torchvision import datasets, transforms

accelerator = Accelerator(mixed_precision="fp16")

# Data
transform = transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize((0.1307,), (0.3081,)),
])
dataset = datasets.MNIST("data", train=True, download=True,
                         transform=transform)
loader = DataLoader(dataset, batch_size=64, shuffle=True)

# Model and optimizer
model = SimpleCNN()  # meme modele que precedemment
optimizer = torch.optim.Adam(model.parameters(), lr=1e-3)
criterion = nn.CrossEntropyLoss()

# Accelerate prepare: wraps model, optimizer, dataloader
model, optimizer, loader = accelerator.prepare(
    model, optimizer, loader
)

# Training loop (identique au code single-GPU !)
for epoch in range(10):
    model.train()
    total_loss = 0.0
    for data, target in loader:
        optimizer.zero_grad()
        output = model(data)
        loss = criterion(output, target)
        accelerator.backward(loss)  # seul changement
        optimizer.step()
        total_loss += loss.item()
```

```
    if accelerator.is_main_process:
        print(f"Epoch {epoch+1}, Loss: {total_loss/len(loader):.4f}")

# Save
accelerator.wait_for_everyone()
if accelerator.is_main_process:
    unwrapped = accelerator.unwrap_model(model)
    torch.save(unwrapped.state_dict(), "model_accelerate.pt")
```

Configuration et lancement Accelerate

```
# Installer
pip install accelerate

# Configuration interactive
accelerate config

# Lancer l'entrainement
accelerate launch --num_processes=4 train_accelerate.py

# Ou avec un fichier de config
accelerate launch --config_file accelerate_config.yaml \
    train_accelerate.py
```

6.5 Entraînement en précision mixte (AMP)

L'entraînement en **précision mixte** utilise FP16 (ou BF16) pour les calculs rapides et FP32 pour les opérations sensibles (accumulation de gradients, normalisation). Cela réduit la mémoire GPU de ∼50% et accélère l'entraînement de 1.5–3× sur les GPU modernes (Ampere, Hopper).

AMP avec PyTorch natif

```
"""Mixed precision training with torch.amp."""
from torch.amp import autocast, GradScaler

scaler = GradScaler("cuda")

for epoch in range(epochs):
    for data, target in loader:
        data, target = data.to(device), target.to(device)

        optimizer.zero_grad()

        # Forward pass in FP16
        with autocast("cuda"):
```

```
            output = model(data)
            loss = criterion(output, target)

        # Backward pass with gradient scaling
        scaler.scale(loss).backward()
        scaler.step(optimizer)
        scaler.update()
```

Remarque 6.3. Le `GradScaler` est nécessaire pour éviter les *underflows* en FP16 : les gradients très petits deviennent zéro en demi-précision. Le scaler multiplie la loss par un facteur élevé, puis divise les gradients avant la mise à jour. Avec BF16 (disponible sur Ampere+), le scaler n'est généralement pas nécessaire car la plage dynamique de BF16 est équivalente à celle de FP32.

6.6 Optimisation des hyperparamètres avec Optuna

Optuna est une bibliothèque d'optimisation d'hyperparamètres qui utilise des algorithmes bayesiens (TPE) pour explorer efficacement l'espace de recherche.

Optimisation avec Optuna

```python
"""Hyperparameter optimization with Optuna."""
import optuna
import torch
import torch.nn as nn
from torch.utils.data import DataLoader
from sklearn.metrics import accuracy_score


def objective(trial: optuna.Trial) -> float:
    """Objective function for Optuna."""
    # Suggest hyperparameters
    lr = trial.suggest_float("lr", 1e-5, 1e-2, log=True)
    n_layers = trial.suggest_int("n_layers", 1, 4)
    hidden_size = trial.suggest_categorical(
        "hidden_size", [64, 128, 256, 512]
    )
    dropout = trial.suggest_float("dropout", 0.0, 0.5)
    batch_size = trial.suggest_categorical(
        "batch_size", [32, 64, 128]
    )

    # Build model dynamically
    layers = []
    in_features = 784  # MNIST
    for i in range(n_layers):
        layers.extend([
            nn.Linear(in_features, hidden_size),
            nn.ReLU(),
```

```
            nn.Dropout(dropout),
        ])
        in_features = hidden_size
    layers.append(nn.Linear(in_features, 10))
    model = nn.Sequential(*layers).to("cuda")

    optimizer = torch.optim.Adam(model.parameters(), lr=lr)
    criterion = nn.CrossEntropyLoss()

    # Train for a few epochs
    for epoch in range(5):
        model.train()
        for data, target in train_loader:
            data = data.view(data.size(0), -1).to("cuda")
            target = target.to("cuda")
            optimizer.zero_grad()
            loss = criterion(model(data), target)
            loss.backward()
            optimizer.step()

        # Pruning: report intermediate value
        val_acc = evaluate(model, val_loader)
        trial.report(val_acc, epoch)
        if trial.should_prune():
            raise optuna.TrialPruned()

    return val_acc


# Run optimization
study = optuna.create_study(
    direction="maximize",
    pruner=optuna.pruners.MedianPruner(n_warmup_steps=2),
)
study.optimize(objective, n_trials=50, timeout=3600)

# Results
print(f"Best trial: {study.best_trial.value:.4f}")
print(f"Best params: {study.best_trial.params}")

# Visualization
optuna.visualization.plot_optimization_history(study)
optuna.visualization.plot_param_importances(study)
```

Bonnes pratiques Optuna

1. **Utiliser le pruning** : arrêter les essais peu prometteurs tôt pour économiser du temps de calcul.
2. **Espaces log-uniformes pour les learning rates** : utiliser

`suggest_float(..., log=True)` pour les paramètres à échelle logarithmique.

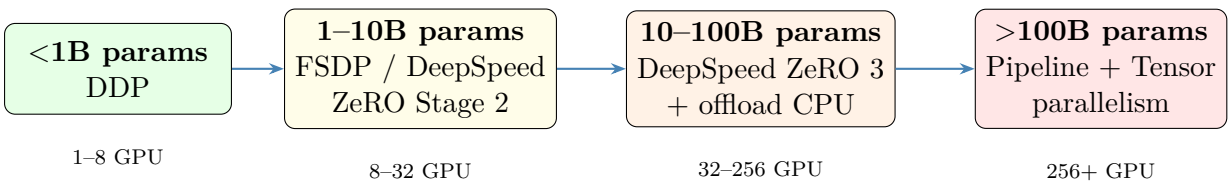
3. **Stocker les résultats** : utiliser un backend persistant (SQLite, MySQL) pour reprendre une étude interrompue.
4. **Paralléliser les essais** : lancer plusieurs workers Optuna en parallèle avec une base de données partagée.

6.7 Comparaison des frameworks d'entraînement distribué

Critère	DDP	FSDP	DeepSpeed	Horovod
Type	Data parallel	Data + model	Data + model	Data parallel
Framework	PyTorch	PyTorch	PyTorch/TF	PyTorch/TF
Sharding poids	Non	Oui	Oui (ZeRO)	Non
Offload CPU	Non	Oui	Oui	Non
Facilité	Moyen	Moyen	Difficile	Facile
Cas d'usage	Modèles <10B	10–100B	10–1000B	Clusters HPC
Optimiseur intégré	Non	Non	Oui	Non
Mixed precision	Manuel	Intégré	Intégré	Manuel

Choisir le bon framework

- **DDP** : défaut pour tout modèle qui tient sur un GPU. Simple et performant.
- **FSDP** : modèles trop grands pour un GPU mais qui tiennent en mémoire agrégée. Solution native PyTorch.
- **DeepSpeed** : modèles très grands (LLMs). ZeRO Stage 3 permet d'entraîner des modèles de centaines de milliards de paramètres. Configuration plus complexe.
- **Horovod** : populaire dans les clusters HPC avec MPI. API simple (`hvd.init()`, `hvd.DistributedOptimizer()`).



6.8 Mini-projet : entraînement distribué

Mini-projet 6 : Entraînement multi-GPU

Mettez en place un pipeline d'entraînement distribué :

1. **Baseline single-GPU** : entraînez un modèle CNN sur CIFAR-10 avec PyTorch standard. Mesurez le temps par époque.
2. **DDP** : adaptez le code pour `DistributedDataParallel`. Comparez le temps par époque avec 2 et 4 GPU.
3. **Mixed precision** : ajoutez l'AMP (`autocast` + `GradScaler`). Mesurez le gain en mémoire et en vitesse.
4. **Accelerate** : réécrivez le code avec `accelerate`. Vérifiez que les résultats sont identiques.
5. **Optuna** : optimisez le learning rate, le batch size et l'architecture avec Optuna (20 trials minimum).
6. **Rapport** : présentez un tableau comparatif : temps par époque, mémoire GPU, accuracy finale pour chaque configuration.

Livrable : dépôt Git avec les scripts, les résultats et le rapport.

6.9 Exercices

Exercice 6.1 (★ — Premier entraînement DDP)**.** Adaptez un script d'entraînement single-GPU existant pour utiliser DDP :

1. Ajoutez `init_process_group` et `DistributedSampler`
2. Wrappez le modèle avec `DistributedDataParallel`
3. Lancez avec `torchrun --nproc_per_node=2`
4. Vérifiez que la loss converge de la même manière qu'en single-GPU

Exercice 6.2 (★★ — Scaling du learning rate)**.** Étudiez l'impact du scaling du learning rate :

1. Entraînez un modèle avec 1 GPU et $\eta = 0.001$, batch size $= 64$
2. Passez à 4 GPU avec batch size effectif $= 256$:
   - Sans scaling du learning rate ($\eta = 0.001$)
   - Avec scaling linéaire ($\eta = 0.004$)
   - Avec scaling linéaire + warmup (5 époques)
3. Tracez les courbes de loss et d'accuracy pour chaque configuration
4. Concluez sur la nécessité du warmup

Exercice 6.3 (★★★ — Pipeline complet avec Optuna et DDP)**.** Construisez un pipeline d'entraînement complet :

1. Écrivez une fonction `objective` Optuna qui entraîne un modèle en DDP
2. Optimisez au moins 5 hyperparamètres (learning rate, architecture, dropout, weight decay, scheduler)
3. Utilisez le pruning médian pour arrêter les essais peu prometteurs
4. Stockez les résultats dans une base SQLite pour la persistance
5. Visualisez les résultats avec les fonctions de visualisation Optuna (importance des paramètres, historique, contour plots)
6. Ré-entraînez le meilleur modèle avec AMP et sauvegardez-le

6.10 Aide-mémoire

Résumé du chapitre 6

Concept	Point clé
Data parallelism	Chaque GPU a le modèle complet, batch divisé
Model parallelism	Modèle partitionné entre GPU
AllReduce	Agrégation des gradients : $\bar{g} = \frac{1}{N}\sum g_k$
DDP	`DistributedDataParallel`, `torchrun`, `DistributedSampler`
Accelerate	`accelerator.prepare()`, `accelerator.backward()`
AMP	`autocast("cuda")`, `GradScaler`, ∼50% mémoire
Linear scaling	$\eta_{\text{dist}} = N \cdot \eta_{\text{base}}$ + warmup
Optuna	`study.optimize(objective, n_trials)`, pruning, TPE
FSDP	Sharding des poids pour modèles 10–100B
DeepSpeed	ZeRO Stages 1–3, offload CPU, modèles >100B

Chapitre 7

Conteneurisation — Docker pour le ML

7.1 Pourquoi les conteneurs ?

Le problème « Ça marche sur ma machine ! » persiste même avec des environnements virtuels. Les causes profondes :

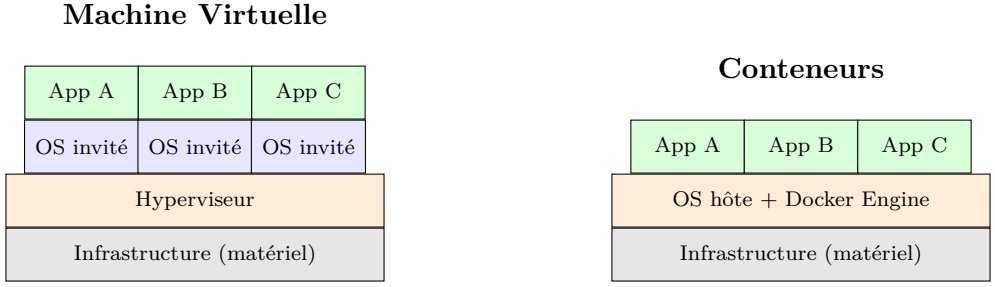
- Différences de système d'exploitation (Ubuntu vs Debian vs Alpine)
- Versions de bibliothèques système (glibc, OpenSSL)
- Drivers GPU et versions de CUDA non alignés
- Variables d'environnement, locales, fuseaux horaires
- Services externes (bases de données, files de messages)

Un environnement virtuel isole les packages Python, mais **pas** le système d'exploitation ni les dépendances système. Les conteneurs résolvent ce problème en encapsulant *tout* l'environnement d'exécution.

Définition 7.1 (Conteneur)**.** Un **conteneur** est une unité logicielle légère et portable qui empaquette le code, les dépendances et la configuration dans un environnement isolé, partageant le noyau du système hôte.

Remarque 7.2. Contrairement à une machine virtuelle, un conteneur ne virtualise pas le matériel : il partage le noyau Linux de l'hôte. Cela le rend beaucoup plus léger (démarrage en secondes, empreinte mémoire réduite).

7.1.1 Conteneurs vs Machines virtuelles



Critère	VM	Conteneur
Démarrage	Minutes	Secondes
Taille	Go	Mo
Isolation	Complète	Niveau processus
Performance	Overhead noyau	Quasi-native
Portabilité	Image lourde	Image légère
GPU passthrough	Complexe	NVIDIA Container Toolkit

7.2 Concepts Docker

Vocabulaire Docker essentiel

Image Modèle en lecture seule contenant le système de fichiers, les dépendances et le code. Analogue à une « classe » en POO.

Conteneur Instance en cours d'exécution d'une image. Analogue à un « objet » instancié depuis la classe.

Dockerfile Fichier texte décrivant les instructions pour construire une image, couche par couche.

Layer (couche) Chaque instruction du Dockerfile crée une couche. Les couches sont cachées et réutilisées.

Registry Dépôt d'images (Docker Hub, GitHub Container Registry, AWS ECR).

Volume Mécanisme de persistance des données en dehors du conteneur.

Tag Étiquette de version d'une image (ex. `python:3.11-slim`).

7.3 Dockerfile pour projets ML

Dockerfile ML complet

```
# Image de base avec Python
FROM python:3.11-slim AS base

# Metadonnees
LABEL maintainer="equipe-ml@example.com"
LABEL description="ML training and inference image"

# Variables d'environnement
ENV PYTHONDONTWRITEBYTECODE=1 \
    PYTHONUNBUFFERED=1 \
    PIP_NO_CACHE_DIR=1

# Dependances systeme
RUN apt-get update && apt-get install -y --no-install-recommends \
```

```
    build-essential \
    curl \
    && rm -rf /var/lib/apt/lists/*

# Creer un utilisateur non-root
RUN useradd --create-home mluser
WORKDIR /home/mluser/app

# Installer les dependances Python d'abord (cache Docker)
COPY requirements.txt .
RUN pip install --no-cache-dir -r requirements.txt

# Copier le code source
COPY src/ ./src/
COPY configs/ ./configs/

# Changer de proprietaire et d'utilisateur
RUN chown -R mluser:mluser /home/mluser/app
USER mluser

# Port pour l'API (si applicable)
EXPOSE 8000

# Commande par defaut
CMD ["python", "src/train.py", "--config", "configs/config.yaml"]
```

Ordre des instructions et cache

Docker met en cache chaque couche. Si une couche change, toutes les couches suivantes sont reconstruites. Règle d'or :

1. Placer les éléments qui changent **rarement** en haut (OS, dépendances système).
2. Copier `requirements.txt` **avant** le code source.
3. Copier le code source en **dernier**.

Ainsi, modifier le code ne déclenche pas la réinstallation des packages.

7.3.1 Dockerfile avec support CUDA

Dockerfile GPU avec CUDA

```
# Image de base NVIDIA CUDA
FROM nvidia/cuda:12.1.0-cudnn8-runtime-ubuntu22.04

ENV DEBIAN_FRONTEND=noninteractive \
    PYTHONDONTWRITEBYTECODE=1 \
    PYTHONUNBUFFERED=1
```

```
# Installer Python
RUN apt-get update && apt-get install -y --no-install-recommends \
    python3.11 python3-pip python3.11-venv \
    && ln -s /usr/bin/python3.11 /usr/bin/python \
    && rm -rf /var/lib/apt/lists/*

WORKDIR /app

COPY requirements.txt .
RUN pip install --no-cache-dir -r requirements.txt

COPY src/ ./src/
COPY configs/ ./configs/

RUN useradd --create-home mluser && chown -R mluser:mluser /app
USER mluser

CMD ["python", "src/train.py"]
```

7.4 Commandes Docker essentielles

Cycle de vie d'un conteneur

```
# Construire une image
docker build -t mon-projet-ml:v1.0 .

# Lancer un conteneur
docker run --name ml-train mon-projet-ml:v1.0

# Lancer en mode interactif
docker run -it mon-projet-ml:v1.0 bash

# Lancer avec un volume (donnees persistantes)
docker run -v $(pwd)/data:/app/data mon-projet-ml:v1.0

# Lancer avec GPU
docker run --gpus all mon-projet-ml:v1.0

# Lister les conteneurs
docker ps -a

# Voir les logs
docker logs ml-train

# Arreter et supprimer
docker stop ml-train && docker rm ml-train

# Lister et supprimer les images
```

```
docker images
docker rmi mon-projet-ml:v1.0
```

7.5 Multi-stage builds

Les **multi-stage builds** permettent de séparer l'environnement de construction de l'environnement d'exécution, réduisant drastiquement la taille de l'image finale.

Multi-stage build pour ML

```
# === Stage 1 : construction ===
FROM python:3.11-slim AS builder

RUN apt-get update && apt-get install -y --no-install-recommends \
    build-essential gfortran libopenblas-dev \
    && rm -rf /var/lib/apt/lists/*

WORKDIR /build
COPY requirements.txt .
RUN pip install --no-cache-dir --prefix=/install -r requirements.txt

# === Stage 2 : execution ===
FROM python:3.11-slim AS runtime

# Copier seulement les packages installes
COPY --from=builder /install /usr/local

RUN useradd --create-home mluser
WORKDIR /home/mluser/app

COPY src/ ./src/
COPY configs/ ./configs/
COPY models/ ./models/

RUN chown -R mluser:mluser /home/mluser/app
USER mluser

EXPOSE 8000
CMD ["python", "src/serve.py"]
```

Approche	Taille image	Sécurité
Single-stage (python :3.11)	∼1.2 Go	Compilateurs présents
Single-stage (python :3.11-slim)	∼800 Mo	Améliorée
Multi-stage (slim → slim)	∼400 Mo	Pas de compilateurs

7.6 Docker Compose pour environnements multi-services

En ML, on a souvent besoin de plusieurs services : l'application, une base de données pour les métadonnées, un serveur MLflow pour le suivi des expériences, etc. **Docker Compose** orchestre ces services dans un seul fichier.

docker-compose.yml — Stack ML complète

```yaml
version: "3.9"

services:
  # --- Application ML ---
  ml-app:
    build:
      context: .
      dockerfile: Dockerfile
    ports:
      - "8000:8000"
    volumes:
      - ./data:/app/data
      - ./models:/app/models
    environment:
      - MLFLOW_TRACKING_URI=http://mlflow:5000
      - DATABASE_URL=postgresql://user:pass@db:5432/mldb
    depends_on:
      - db
      - mlflow
    deploy:
      resources:
        reservations:
          devices:
            - capabilities: [gpu]

  # --- Base de donnees PostgreSQL ---
  db:
    image: postgres:16-alpine
    environment:
      POSTGRES_USER: user
      POSTGRES_PASSWORD: pass
      POSTGRES_DB: mldb
    volumes:
      - pgdata:/var/lib/postgresql/data
    ports:
      - "5432:5432"

  # --- MLflow Tracking Server ---
  mlflow:
    image: ghcr.io/mlflow/mlflow:v2.10.0
    command: >
      mlflow server
      --backend-store-uri postgresql://user:pass@db:5432/mldb
```

```
      --default-artifact-root /mlflow/artifacts
      --host 0.0.0.0
      --port 5000
    ports:
      - "5000:5000"
    volumes:
      - mlflow-artifacts:/mlflow/artifacts
    depends_on:
      - db

volumes:
  pgdata:
  mlflow-artifacts:
```

Commandes Docker Compose

```
# Demarrer tous les services
docker compose up -d

# Voir les logs de tous les services
docker compose logs -f

# Voir les logs d'un service specifique
docker compose logs -f ml-app

# Arreter tous les services
docker compose down

# Arreter et supprimer les volumes
docker compose down -v

# Reconstruire les images
docker compose build --no-cache

# Lancer une commande dans un service
docker compose exec ml-app python src/train.py
```

7.7 Le fichier .dockerignore

Le fichier `.dockerignore` empêche l'envoi de fichiers inutiles au démon Docker lors du build. Sans lui, Docker copie *tout* le contexte, y compris les données, les modèles et les environnements virtuels.

.dockerignore pour projet ML

```
# Environnements virtuels
.venv/
```

```
venv/
__pycache__/
*.pyc

# Donnees volumineuses
data/raw/
data/processed/
*.csv
*.parquet

# Modeles sauvegardes
models/*.joblib
models/*.pkl
*.onnx

# Git et IDE
.git/
.gitignore
.vscode/
.idea/

# Notebooks
*.ipynb
.ipynb_checkpoints/

# Docker
Dockerfile
docker-compose.yml
.dockerignore

# Documentation
*.md
docs/
```

7.8 Bonnes pratiques Docker pour le ML

Règles d'or Docker en ML

1. **Images légères** : utiliser `python:3.11-slim` ou `alpine`, pas `python:3.11` (économie de 600 Mo).
2. **Utilisateur non-root** : toujours créer et utiliser un utilisateur dédié (`USER mluser`).
3. **Exploiter le cache** : copier `requirements.txt` avant le code source.
4. **Multi-stage builds** : séparer construction et exécution.
5. **.dockerignore** : exclure données, modèles, `.git/`.

6. **Tags explicites** : `python:3.11-slim`, jamais `python:latest`.
7. **Un processus par conteneur** : ne pas lancer la base de données et l'application dans le même conteneur.
8. **Healthchecks** : ajouter `HEALTHCHECK` pour les services de serving.

Anti-patterns Docker en ML

- **Images géantes** : inclure les compilateurs, les données d'entraînement et les checkpoints dans l'image (résultat : 10+ Go).
- **Exécution en root** : lancer le conteneur en tant que root expose l'hôte en cas de vulnérabilité.
- **Secrets dans l'image** : mettre des clés API, mots de passe ou tokens dans le Dockerfile ou dans les couches de l'image. Utiliser plutôt des variables d'environnement ou Docker Secrets.
- **Tag `latest`** : ne permet pas de reproduire un build. Toujours épingler une version précise.
- **Ignorer le .dockerignore** : envoyer des Go de données au démon Docker à chaque build.
- **Un seul gros conteneur** : mettre l'app, la BDD et MLflow dans le même conteneur.
- **Pas de healthcheck** : impossible de savoir si le service est réellement fonctionnel.

7.9 Registries et publication d'images

Publier une image sur Docker Hub

```
# Se connecter
docker login

# Taguer l'image
docker tag mon-projet-ml:v1.0 monuser/mon-projet-ml:v1.0

# Pousser l'image
docker push monuser/mon-projet-ml:v1.0

# Tirer l'image (sur une autre machine)
docker pull monuser/mon-projet-ml:v1.0
```

Utiliser GitHub Container Registry

```
# Se connecter avec un token GitHub
echo $GITHUB_TOKEN | docker login ghcr.io -u USERNAME --password-stdin

# Taguer et pousser
docker tag mon-projet-ml:v1.0 ghcr.io/USERNAME/mon-projet-ml:v1.0
docker push ghcr.io/USERNAME/mon-projet-ml:v1.0
```

7.10 Mini-projet : conteneuriser le projet ML du cours

Mini-projet 7 : Conteneurisation complète

En reprenant le projet ML structuré des chapitres précédents :

1. Écrivez un `Dockerfile` multi-stage pour le projet :
   - Stage 1 : installer les dépendances et compiler
   - Stage 2 : copier uniquement les packages et le code
2. Créez un `.dockerignore` adapté.
3. Construisez l'image et vérifiez sa taille (`docker images`).
4. Lancez un entraînement dans le conteneur avec un volume pour les données : `docker run -v $(pwd)/data:/app/data ...`
5. Écrivez un `docker-compose.yml` avec trois services :
   - `ml-app` : votre application ML
   - `db` : PostgreSQL pour MLflow
   - `mlflow` : serveur de suivi des expériences
6. Vérifiez que `docker compose up` démarre tout correctement et que MLflow est accessible sur `http://localhost:5000`.

7.11 Exercices

Exercice 7.1 (★ — Premier Dockerfile)**.** Écrivez un `Dockerfile` pour un script Python simple qui charge un dataset CSV avec `pandas` et affiche des statistiques descriptives. Construisez l'image, lancez le conteneur et vérifiez la sortie. Comparez la taille de l'image avec `python:3.11` vs `python:3.11-slim`.

Exercice 7.2 (★★ — Optimisation d'image)**.** Partez d'un Dockerfile naïf (image `python:3.11`, pas de `.dockerignore`, copie de tout le répertoire, exécution en root). Améliorez-le itérativement :

1. Passez à `python:3.11-slim` — notez la réduction de taille

2. Ajoutez un `.dockerignore` — mesurez le temps de build
3. Réordonnez les instructions pour exploiter le cache
4. Passez en multi-stage build
5. Ajoutez un utilisateur non-root

Documentez la taille de l'image et le temps de build à chaque étape.

Exercice 7.3 (★★★ — Stack ML complète avec Docker Compose)**.** Créez un `docker-compose.yml` complet pour un projet ML incluant :

1. Un service d'entraînement (GPU si disponible)
2. Un service d'inférence FastAPI (port 8000)
3. Un serveur MLflow (port 5000)
4. Une base de données PostgreSQL
5. Un dashboard Streamlit (port 8501)
6. Des healthchecks pour chaque service
7. Des volumes pour la persistance des données et des modèles

Vérifiez que tous les services communiquent correctement : l'entraînement loggue dans MLflow, l'API charge le modèle depuis le volume partagé, et le dashboard interroge l'API.

7.12 Aide-mémoire

Résumé du chapitre 7

Concept	Commande / Point clé
Construire	`docker build -t name:tag .`
Lancer	`docker run -v data:/app/data name:tag`
GPU	`docker run --gpus all name:tag`
Multi-stage	`FROM ... AS builder` puis `COPY --from=builder`
Compose up	`docker compose up -d`
Compose down	`docker compose down -v`
Cache optimal	`requirements.txt` avant code source
Sécurité	`USER mluser`, jamais de secrets dans l'image
Image légère	`python:3.11-slim` + multi-stage

Chapitre 8

Déploiement de Modèles — REST APIs, FastAPI, Streamlit

8.1 Patterns de déploiement

Un modèle entraîné n'a de valeur que s'il est accessible aux utilisateurs ou aux systèmes aval. Trois grands patterns de déploiement existent :

Définition 8.1 (Inférence batch)**.** L'**inférence batch** consiste à exécuter les prédictions sur un lot de données à intervalles réguliers (par exemple, chaque nuit). Les résultats sont stockés dans une base de données ou un fichier.

Définition 8.2 (Inférence en ligne)**.** L'**inférence en ligne** (ou temps réel) répond à des requêtes individuelles via une API REST ou gRPC, avec une latence typique de quelques millisecondes à quelques secondes.

Définition 8.3 (Inférence en streaming)**.** L'**inférence en streaming** traite un flux continu de données (Kafka, Flink) et produit des prédictions en continu, adaptée aux cas où les données arrivent en temps réel.

Critère	**Batch**	**En ligne**	**Streaming**
Latence	Minutes–heures	Millisecondes	Secondes
Débit	Très élevé	Moyen	Élevé
Complexité	Faible	Moyenne	Élevée
Cas d'usage	Scoring nocturne	API web/mobile	Détection fraude
Infra typique	Cron + Spark	FastAPI + K8s	Kafka + Flink

Remarque 8.4. La majorité des projets ML en entreprise commencent par du batch, puis évoluent vers du temps réel quand le besoin métier le justifie. Ne sur-ingéniez pas dès le départ.

8.2 Concepts REST API pour le ML

Une API REST expose le modèle via des endpoints HTTP. Les conventions :

Endpoints typiques d'une API ML

Méthode	Endpoint	Description
GET	/health	Vérification que le service est actif
GET	/model/info	Métadonnées du modèle (version, métriques)
POST	/predict	Prédiction sur une ou plusieurs instances
POST	/predict/batch	Prédiction batch (fichier CSV)

8.3 FastAPI : servir un modèle ML

FastAPI est un framework Python moderne, rapide (basé sur Starlette et Pydantic), avec documentation automatique OpenAPI et validation des données intégrée.

8.3.1 Schémas Pydantic

schemas.py — Schémas de requête et réponse

```python
"""Pydantic schemas for request/response validation."""
from pydantic import BaseModel, Field


class PredictionRequest(BaseModel):
    """Schema for a single prediction request."""
    sepal_length: float = Field(..., gt=0, description="Sepal length in
    ↪ cm")
    sepal_width: float = Field(..., gt=0, description="Sepal width in
    ↪ cm")
    petal_length: float = Field(..., gt=0, description="Petal length in
    ↪ cm")
    petal_width: float = Field(..., gt=0, description="Petal width in
    ↪ cm")

    model_config = {"json_schema_extra": {
        "examples": [
            {
                "sepal_length": 5.1,
                "sepal_width": 3.5,
                "petal_length": 1.4,
                "petal_width": 0.2,
            }
        ]
    }}


class PredictionResponse(BaseModel):
```

```
    """Schema for prediction response."""
    prediction: int = Field(..., description="Predicted class (0, 1, or
    ↪  2)")
    class_name: str = Field(..., description="Human-readable class name")
    probabilities: list[float] = Field(
        ..., description="Probability for each class"
    )


class HealthResponse(BaseModel):
    """Schema for health check response."""
    status: str
    model_loaded: bool
    model_version: str
```

8.3.2 Application FastAPI complète

app.py — API de serving ML complète

```
"""FastAPI ML serving application."""
import joblib
import numpy as np
from contextlib import asynccontextmanager
from fastapi import FastAPI, HTTPException
from schemas import PredictionRequest, PredictionResponse, HealthResponse

# Variable globale pour le modele
model = None
MODEL_PATH = "models/rf_model.joblib"
MODEL_VERSION = "1.0.0"
CLASS_NAMES = ["setosa", "versicolor", "virginica"]


@asynccontextmanager
async def lifespan(app: FastAPI):
    """Load model on startup, cleanup on shutdown."""
    global model
    try:
        model = joblib.load(MODEL_PATH)
        print(f"Model loaded from {MODEL_PATH}")
    except FileNotFoundError:
        print(f"WARNING: Model not found at {MODEL_PATH}")
    yield
    model = None
    print("Model unloaded")


app = FastAPI(
    title="ML Prediction API",
```

```
    description="API de prediction pour le modele Iris",
    version=MODEL_VERSION,
    lifespan=lifespan,
)


@app.get("/health", response_model=HealthResponse)
async def health_check():
    """Health check endpoint."""
    return HealthResponse(
        status="healthy" if model is not None else "degraded",
        model_loaded=model is not None,
        model_version=MODEL_VERSION,
    )


@app.post("/predict", response_model=PredictionResponse)
async def predict(request: PredictionRequest):
    """Generate prediction for a single instance."""
    if model is None:
        raise HTTPException(status_code=503, detail="Model not loaded")

    features = np.array([[
        request.sepal_length,
        request.sepal_width,
        request.petal_length,
        request.petal_width,
    ]])

    prediction = int(model.predict(features)[0])
    probabilities = model.predict_proba(features)[0].tolist()

    return PredictionResponse(
        prediction=prediction,
        class_name=CLASS_NAMES[prediction],
        probabilities=[round(p, 4) for p in probabilities],
    )
```

Lancer et tester l'API

```
# Installer FastAPI et uvicorn
pip install fastapi uvicorn

# Lancer le serveur
uvicorn app:app --host 0.0.0.0 --port 8000 --reload

# Tester avec curl
curl -X POST http://localhost:8000/predict \
  -H "Content-Type: application/json" \
  -d '{"sepal_length": 5.1, "sepal_width": 3.5,
```

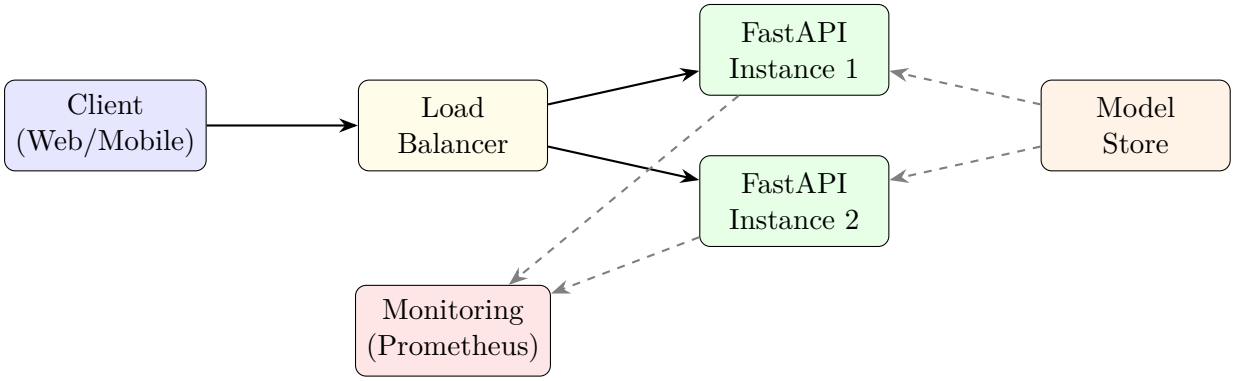
```
       "petal_length": 1.4, "petal_width": 0.2}'

# Documentation automatique
# Swagger UI : http://localhost:8000/docs
# ReDoc :      http://localhost:8000/redoc
```

Sortie

```
{
  "prediction": 0,
  "class_name": "setosa",
  "probabilities": [0.97, 0.02, 0.01]
}
```

8.4 Architecture de déploiement



8.5 Streamlit : construire un démo UI

Streamlit permet de créer rapidement une interface web interactive pour démontrer un modèle ML, sans connaître le développement frontend.

streamlit_app.py — Démo interactive

```
"""Streamlit demo for the Iris classification model."""
import streamlit as st
import requests
import json

API_URL = "http://localhost:8000"

st.set_page_config(page_title="Iris Classifier", page_icon=" ")
st.title("Classification Iris")
st.markdown("Entrez les mesures de la fleur pour obtenir une
↪  prediction.")
```

```python
# Sidebar pour les parametres
st.sidebar.header("Parametres de la fleur")
sepal_length = st.sidebar.slider("Longueur sepale (cm)", 4.0, 8.0, 5.1)
sepal_width = st.sidebar.slider("Largeur sepale (cm)", 2.0, 4.5, 3.5)
petal_length = st.sidebar.slider("Longueur petale (cm)", 1.0, 7.0, 1.4)
petal_width = st.sidebar.slider("Largeur petale (cm)", 0.1, 2.5, 0.2)

# Bouton de prediction
if st.button("Predire"):
    payload = {
        "sepal_length": sepal_length,
        "sepal_width": sepal_width,
        "petal_length": petal_length,
        "petal_width": petal_width,
    }

    try:
        response = requests.post(f"{API_URL}/predict", json=payload)
        response.raise_for_status()
        result = response.json()

        st.success(f"Prediction : **{result['class_name']}**")

        # Afficher les probabilites
        st.subheader("Probabilites par classe")
        class_names = ["setosa", "versicolor", "virginica"]
        for name, prob in zip(class_names, result["probabilities"]):
            st.progress(prob, text=f"{name}: {prob:.1%}")

    except requests.exceptions.ConnectionError:
        st.error("Impossible de se connecter a l'API. "
                 "Verifiez que le serveur tourne.")

# Health check
st.sidebar.markdown("---")
if st.sidebar.button("Verifier l'API"):
    try:
        health = requests.get(f"{API_URL}/health").json()
        st.sidebar.json(health)
    except requests.exceptions.ConnectionError:
        st.sidebar.error("API indisponible")
```

Lancer le dashboard Streamlit

```bash
# Installer Streamlit
pip install streamlit requests

# Lancer l'application
streamlit run streamlit_app.py --server.port 8501
```

```
# Accessible sur http://localhost:8501
```

8.6 Sérialisation de modèles

Le choix du format de sérialisation affecte la portabilité, la vitesse d'inférence et la compatibilité inter-frameworks.

Formats de sérialisation

Format	Avantages	Inconvénients	Usage
joblib/pickle	Simple, natif Python	Python uniquement, vulnérable	Prototypage, scikit-learn
ONNX	Multi-framework, optimisé	Pas tous les opérateurs	Production cross-platform
TorchScript	Optimisé PyTorch, C++	PyTorch uniquement	Production PyTorch
SavedModel	Écosystème TF complet	TensorFlow uniquement	TF Serving, TFLite

Export de modèles dans différents formats

```python
"""Model serialization examples."""
import joblib
import torch
import onnx
from skl2onnx import convert_sklearn
from skl2onnx.common.data_types import FloatTensorType

# --- joblib (scikit-learn) ---
joblib.dump(model, "model.joblib")
model = joblib.load("model.joblib")

# --- ONNX (scikit-learn) ---
initial_type = [("features", FloatTensorType([None, 4]))]
onnx_model = convert_sklearn(model, initial_types=initial_type)
onnx.save(onnx_model, "model.onnx")

# Inference ONNX
import onnxruntime as ort
session = ort.InferenceSession("model.onnx")
input_name = session.get_inputs()[0].name
result = session.run(None, {input_name: features.astype("float32")})

# --- TorchScript (PyTorch) ---
scripted_model = torch.jit.script(pytorch_model)
scripted_model.save("model.pt")
```

```
loaded = torch.jit.load("model.pt")
output = loaded(torch.tensor(features))
```

Sécurité de pickle/joblib

Les fichiers `.pkl` et `.joblib` peuvent exécuter du code arbitraire au chargement. Ne chargez **jamais** un modèle sérialisé provenant d'une source non fiable. En production, préférez ONNX ou TorchScript.

8.7 Comparaison des frameworks de serving

Critère	FastAPI	Flask	TorchServe	Triton
Async natif	Oui	Non	Oui	Oui
Validation auto	Pydantic	Manuelle	Manuelle	Manuelle
Doc OpenAPI	Automatique	Manuelle	Non	Non
Multi-framework	Oui	Oui	PyTorch	Tous
GPU optimisé	Manuel	Manuel	Oui	Oui
Batching	Manuel	Manuel	Automatique	Automatique
Monitoring	Manuel	Manuel	Intégré	Intégré
Complexité	Faible	Faible	Moyenne	Élevée
Cas d'usage	API générale	Legacy	PyTorch prod	Haute perf

Remarque 8.5. Pour débuter, FastAPI est le meilleur compromis entre simplicité et performance. TorchServe et Triton sont indiqués pour les déploiements à haute performance nécessitant du batching dynamique et une optimisation GPU avancée.

8.8 Dockerfile pour le déploiement

Dockerfile pour l'API + Streamlit

```
# === API FastAPI ===
FROM python:3.11-slim AS api

WORKDIR /app
COPY requirements-api.txt .
RUN pip install --no-cache-dir -r requirements-api.txt

COPY src/ ./src/
COPY models/ ./models/
COPY schemas.py app.py ./

RUN useradd --create-home mluser && chown -R mluser:mluser /app
USER mluser

EXPOSE 8000
```

```
HEALTHCHECK --interval=30s --timeout=5s --retries=3 \
  CMD curl -f http://localhost:8000/health || exit 1

CMD ["uvicorn", "app:app", "--host", "0.0.0.0", "--port", "8000"]
```

docker-compose.yml pour API + Streamlit

```
version: "3.9"

services:
  api:
    build:
      context: .
      dockerfile: Dockerfile
      target: api
    ports:
      - "8000:8000"
    volumes:
      - ./models:/app/models
    healthcheck:
      test: ["CMD", "curl", "-f", "http://localhost:8000/health"]
      interval: 30s
      timeout: 5s
      retries: 3

  streamlit:
    build:
      context: .
      dockerfile: Dockerfile.streamlit
    ports:
      - "8501:8501"
    environment:
      - API_URL=http://api:8000
    depends_on:
      api:
        condition: service_healthy
```

8.9 Bonnes pratiques de déploiement

Règles d'or du déploiement ML

1. **Healthcheck obligatoire** : le endpoint `/health` doit vérifier que le modèle est chargé et fonctionnel.
2. **Versionner les modèles** : inclure la version dans l'endpoint `/model/info` et dans les réponses.
3. **Valider les entrées** : utiliser Pydantic pour rejeter les requêtes malformées

avant l'inférence.

4. **Logging structuré** : chaque prédiction doit être loggée avec les features d'entrée, la prédiction et la latence.
5. **Timeout et retry** : configurer des timeouts côté client et serveur.
6. **Séparer API et UI** : FastAPI et Streamlit dans des conteneurs séparés.
7. **Load testing** : utiliser `locust` ou `wrk` pour mesurer la capacité avant la mise en production.

Anti-patterns de déploiement

- Charger le modèle à chaque requête (utiliser le lifespan).
- Pas de validation des entrées (crash sur données inattendues).
- Retourner des erreurs génériques (toujours « 500 Internal Server Error ») sans détails utiles.
- Pas de healthcheck (le load balancer envoie du trafic à un service mort).
- Déployer sans tests d'intégration de l'API.
- Ignorer la latence : ne pas mesurer le temps d'inférence.

8.10 Mini-projet : déployer le modèle du cours

Mini-projet 8 : API + Démo Streamlit

En reprenant le modèle entraîné dans les chapitres précédents :

1. **Sérialisation** : sauvegardez le modèle en `.joblib` et en ONNX. Comparez les tailles.
2. **API FastAPI** :
   - Créez les schémas Pydantic adaptés à votre modèle
   - Implémentez `/health`, `/model/info` et `/predict`
   - Ajoutez un logging de chaque prédiction (features, résultat, latence)
3. **Démo Streamlit** : créez un dashboard interactif qui :
   - Permet de saisir les features via des sliders
   - Affiche la prédiction et les probabilités
   - Affiche un historique des dernières prédictions
4. **Conteneurisation** : créez un `docker-compose.yml` avec les services `api` et `streamlit`.

5. **Tests** : écrivez des tests d'intégration avec `httpx` ou `TestClient` de FastAPI.
6. **Load testing** : mesurez la latence et le débit avec `locust` (objectif : <100 ms p95 pour une instance).

8.11 Exercices

Exercice 8.1 (★ — Première API FastAPI)**.** Créez une API FastAPI minimaliste qui charge un modèle scikit-learn entraîné et expose un endpoint `/predict`. Testez-la avec `curl` et la documentation Swagger automatique (`http://localhost:8000/docs`).

Exercice 8.2 (★★ — Validation et gestion d'erreurs)**.** Améliorez l'API de l'exercice précédent :

1. Ajoutez des schémas Pydantic avec des contraintes de validation (bornes min/max, types stricts)
2. Implémentez un endpoint `/predict/batch` qui accepte une liste d'instances
3. Ajoutez une gestion d'erreurs complète : modèle non chargé (503), données invalides (422), erreur interne (500) avec messages clairs
4. Ajoutez un middleware de logging qui enregistre chaque requête (méthode, endpoint, latence, code de retour)
5. Écrivez des tests unitaires avec `TestClient` de FastAPI

Exercice 8.3 (★★★ — Déploiement complet avec ONNX)**.** Réalisez un déploiement production-ready :

1. Exportez le modèle en ONNX. Comparez la latence d'inférence entre joblib et ONNX Runtime
2. Créez une API FastAPI avec support ONNX (`onnxruntime`)
3. Ajoutez un dashboard Streamlit connecté à l'API
4. Conteneurisez le tout avec Docker Compose (API + Streamlit)
5. Rédigez un load test avec `locust` : mesurez la latence p50, p95, p99 et le débit maximal
6. Configurez un reverse proxy Nginx devant l'API
7. Documentez l'architecture dans un schéma (TikZ ou draw.io)

8.12 Aide-mémoire

Résumé du chapitre 8

Concept	Point clé
Patterns	Batch (cron) vs En ligne (API) vs Streaming (Kafka)
FastAPI	`uvicorn app:app --port 8000`, Pydantic, async
Streamlit	`streamlit run app.py`, sliders, `st.button`
Sérialisation	joblib (proto), ONNX (prod), TorchScript (PyTorch)
Healthcheck	`GET /health` avec statut modèle
Validation	Pydantic `BaseModel` avec `Field` contraintes
Serving avancé	TorchServe (PyTorch), Triton (multi-framework)
Anti-pattern	Charger le modèle par requête, pas de healthcheck

Chapitre 9

CI/CD pour le Machine Learning

9.1 Introduction

Dans le développement logiciel traditionnel, les pratiques de CI/CD (Intégration Continue / Déploiement Continu) ont révolutionné la manière dont les équipes livrent du code. En Machine Learning, ces pratiques deviennent encore plus critiques : il ne s'agit plus seulement de tester du code, mais également de valider des données, des modèles et des pipelines d'entraînement complets.

Définition 9.1 (Intégration Continue (CI))**.** L'intégration continue est une pratique de développement où les développeurs fusionnent fréquemment leurs modifications dans une branche principale. Chaque fusion déclenche une compilation et des tests automatisés pour détecter les erreurs le plus tôt possible.

Définition 9.2 (Livraison Continue (CD))**.** La livraison continue étend la CI en automatisant le processus de mise en production. On distingue :

- **Continuous Delivery** : le code est prêt à être déployé à tout moment, mais le déploiement reste manuel.
- **Continuous Deployment** : chaque modification validée est automatiquement déployée en production.

9.2 CI/CD pour le ML : qu'est-ce qui change ?

Le ML introduit des défis supplémentaires par rapport au développement logiciel classique.

Spécificités du CI/CD en ML

Contrairement au logiciel traditionnel, un pipeline ML doit valider :

1. **Le code** – qualité, linting, typage
2. **Les données** – schéma, distribution, intégrité
3. **Le modèle** – performance, équité, robustesse
4. **L'infrastructure** – reproductibilité, ressources

Un changement de données peut casser un modèle *sans qu'aucune ligne de code n'ait changé.*

Différences CI/CD classique vs ML

Aspect	CI/CD classique	CI/CD ML
Entrée	Code source	Code + Données + Modèle
Tests	Unitaires, intégration	+ validation données, perf. modèle
Artefact	Binaire, image Docker	+ modèle sérialisé, métriques
Déclencheur	Commit, PR	+ nouveau dataset, drift détecté
Validation	Tests passent	+ seuils de performance atteints
Déploiement	Blue/Green, Canary	+ A/B testing, Shadow mode

9.2.1 Stratégies de déploiement ML

Définition 9.3 (A/B Testing pour les modèles)**.** L'A/B testing en ML consiste à exposer simultanément deux versions d'un modèle à des sous-ensembles d'utilisateurs, puis à comparer les métriques métier (taux de clic, revenu, satisfaction) pour déterminer quelle version est supérieure. Cette approche nécessite une infrastructure capable de router le trafic et de collecter des métriques en temps réel.

Définition 9.4 (Shadow Deployment)**.** Le *shadow deployment* (ou *dark launch*) consiste à déployer un nouveau modèle en parallèle du modèle en production. Le nouveau modèle reçoit le même trafic, mais ses prédictions ne sont pas exposées aux utilisateurs. Cela permet de comparer les performances sans risque.

9.3 GitHub Actions : workflow complet pour un projet ML

GitHub Actions est l'outil de CI/CD intégré à GitHub. Il utilise des fichiers YAML pour définir des workflows déclenchés par des événements (push, pull request, etc.).

Workflow CI/CD complet pour un projet ML

```yaml
# .github/workflows/ml-pipeline.yml
name: ML Pipeline CI/CD

on:
  push:
    branches: [main, develop]
  pull_request:
    branches: [main]

env:
  PYTHON_VERSION: "3.11"
  DVC_REMOTE: s3://my-bucket/dvc-store
```

```yaml
jobs:
  lint-and-format:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v4
      - uses: actions/setup-python@v5
        with:
          python-version: ${{ env.PYTHON_VERSION }}
      - name: Install linting tools
        run: pip install ruff black mypy
      - name: Check formatting (Black)
        run: black --check src/ tests/
      - name: Lint (Ruff)
        run: ruff check src/ tests/
      - name: Type checking (mypy)
        run: mypy src/ --ignore-missing-imports

  test:
    runs-on: ubuntu-latest
    needs: lint-and-format
    steps:
      - uses: actions/checkout@v4
      - uses: actions/setup-python@v5
        with:
          python-version: ${{ env.PYTHON_VERSION }}
      - name: Install dependencies
        run: |
          pip install -r requirements.txt
          pip install pytest pytest-cov
      - name: Run unit tests
        run: pytest tests/unit/ -v --cov=src/ --cov-report=xml
      - name: Run integration tests
        run: pytest tests/integration/ -v
      - name: Upload coverage
        uses: codecov/codecov-action@v3

  train-and-validate:
    runs-on: ubuntu-latest
    needs: test
    steps:
      - uses: actions/checkout@v4
      - uses: actions/setup-python@v5
        with:
          python-version: ${{ env.PYTHON_VERSION }}
      - name: Install DVC
        run: pip install dvc[s3]
      - name: Pull data with DVC
        run: dvc pull
        env:
          AWS_ACCESS_KEY_ID: ${{ secrets.AWS_ACCESS_KEY_ID }}
```

```yaml
          AWS_SECRET_ACCESS_KEY: ${{ secrets.AWS_SECRET_KEY }}
      - name: Reproduce pipeline
        run: dvc repro
      - name: Validate model performance
        run: python scripts/validate_model.py
          --min-accuracy 0.85
          --max-latency-ms 100
          --check-drift

  deploy:
    runs-on: ubuntu-latest
    needs: train-and-validate
    if: github.ref == 'refs/heads/main'
    steps:
      - uses: actions/checkout@v4
      - name: Build Docker image
        run: docker build -t ml-model:${{ github.sha }} .
      - name: Push to registry
        run: |
          docker tag ml-model:${{ github.sha }} \
            ghcr.io/${{ github.repository }}/model:latest
          docker push ghcr.io/${{ github.repository }}/model:latest
      - name: Deploy to staging
        run: |
          kubectl set image deployment/ml-model \
            model=ghcr.io/${{ github.repository }}/model:latest
```

9.4 Pre-commit hooks pour la qualité du code

Les *pre-commit hooks* permettent d'exécuter des vérifications automatiques avant chaque commit, garantissant que le code qui entre dans le dépôt respecte les standards de qualité.

Configuration .pre-commit-config.yaml

```yaml
repos:
  - repo: https://github.com/psf/black
    rev: 24.4.2
    hooks:
      - id: black
        language_version: python3.11

  - repo: https://github.com/astral-sh/ruff-pre-commit
    rev: v0.4.4
    hooks:
      - id: ruff
        args: [--fix, --exit-non-zero-on-fix]
```

```yaml
  - repo: https://github.com/pre-commit/mirrors-mypy
    rev: v1.10.0
    hooks:
      - id: mypy
        additional_dependencies: [numpy, pandas-stubs]

  - repo: https://github.com/pre-commit/pre-commit-hooks
    rev: v4.6.0
    hooks:
      - id: check-yaml
      - id: check-json
      - id: check-added-large-files
        args: [--maxkb=500]
      - id: trailing-whitespace
      - id: end-of-file-fixer
```

Installation et utilisation

```bash
# Installation
pip install pre-commit

# Initialiser les hooks dans le depot
pre-commit install

# Executer sur tous les fichiers (premiere fois)
pre-commit run --all-files
```

Les hooks s'exécutent automatiquement à chaque `git commit`. Si un hook échoue, le commit est bloqué jusqu'à correction.

Remarque 9.5. L'option `--maxkb=500` dans `check-added-large-files` empêche l'ajout accidentel de fichiers volumineux (datasets, modèles sérialisés) dans le dépôt Git. Utilisez DVC pour versionner ces fichiers à la place.

9.5 Validation de modèle dans la CI

La validation automatisée du modèle est l'étape la plus spécifique au ML dans un pipeline CI/CD.

Script de validation de modèle

```python
# scripts/validate_model.py
import argparse
import json
import sys
from pathlib import Path

import joblib
```

```python
import numpy as np
from scipy import stats
from sklearn.metrics import accuracy_score, f1_score


def load_metrics(path: str = "metrics.json") -> dict:
    """Charge les metriques du modele entraine."""
    with open(path) as f:
        return json.load(f)


def check_performance(metrics: dict, min_accuracy: float) -> bool:
    """Verifie que les metriques depassent les seuils."""
    accuracy = metrics.get("accuracy", 0.0)
    if accuracy < min_accuracy:
        print(f"ECHEC: accuracy={accuracy:.4f} < {min_accuracy}")
        return False
    print(f"OK: accuracy={accuracy:.4f} >= {min_accuracy}")
    return True


def check_data_drift(
    reference_path: str, current_path: str, threshold: float = 0.05
) -> bool:
    """Detecte le drift via le test de Kolmogorov-Smirnov."""
    ref = np.load(reference_path)
    cur = np.load(current_path)

    drift_detected = False
    for i in range(ref.shape[1]):
        stat, p_value = stats.ks_2samp(ref[:, i], cur[:, i])
        if p_value < threshold:
            print(f"DRIFT feature {i}: KS={stat:.4f}, p={p_value:.6f}")
            drift_detected = True

    return not drift_detected


def check_latency(model_path: str, sample_path: str,
                  max_latency_ms: float) -> bool:
    """Verifie la latence d'inference."""
    import time
    model = joblib.load(model_path)
    sample = np.load(sample_path)[:100]

    start = time.perf_counter()
    for x in sample:
        model.predict(x.reshape(1, -1))
    elapsed_ms = (time.perf_counter() - start) / len(sample) * 1000

    if elapsed_ms > max_latency_ms:
```

```
        print(f"ECHEC: latence={elapsed_ms:.2f}ms > {max_latency_ms}ms")
        return False
    print(f"OK: latence={elapsed_ms:.2f}ms <= {max_latency_ms}ms")
    return True


if __name__ == "__main__":
    parser = argparse.ArgumentParser()
    parser.add_argument("--min-accuracy", type=float, default=0.85)
    parser.add_argument("--max-latency-ms", type=float, default=100)
    parser.add_argument("--check-drift", action="store_true")
    args = parser.parse_args()

    metrics = load_metrics()
    checks = [check_performance(metrics, args.min_accuracy)]

    if args.check_drift:
        checks.append(check_data_drift(
            "data/reference.npy", "data/current.npy"
        ))

    checks.append(check_latency(
        "models/model.pkl", "data/current.npy", args.max_latency_ms
    ))

    if not all(checks):
        print("\nValidation ECHOUEE -- deploiement bloque.")
        sys.exit(1)
    print("\nValidation REUSSIE -- pret pour le deploiement.")
```

9.6 DVC dans la CI : reproduire les pipelines automatiquement

DVC (Data Version Control) permet de versionner les données et de définir des pipelines reproductibles. Dans un contexte CI, DVC garantit que le pipeline est exécuté de manière déterministe.

Pipeline DVC `dvc.yaml`

```
stages:
  preprocess:
    cmd: python src/preprocess.py
    deps:
      - src/preprocess.py
      - data/raw/
    outs:
      - data/processed/
    params:
```

```yaml
      - preprocess.test_size
      - preprocess.seed

  train:
    cmd: python src/train.py
    deps:
      - src/train.py
      - data/processed/
    outs:
      - models/model.pkl
    params:
      - train.n_estimators
      - train.learning_rate
    metrics:
      - metrics.json:
          cache: false

  evaluate:
    cmd: python src/evaluate.py
    deps:
      - src/evaluate.py
      - models/model.pkl
      - data/processed/test.csv
    metrics:
      - eval_metrics.json:
          cache: false
    plots:
      - plots/confusion_matrix.csv:
          x: predicted
          y: actual
```

Commandes DVC dans la CI

```
# Reproduire le pipeline (seules les etapes modifiees)
dvc repro

# Comparer les metriques avec la branche principale
dvc metrics diff main

# Afficher les parametres
dvc params diff main
```

La commande `dvc repro` n'exécute que les étapes dont les dépendances ont changé, ce qui accélère considérablement les pipelines CI.

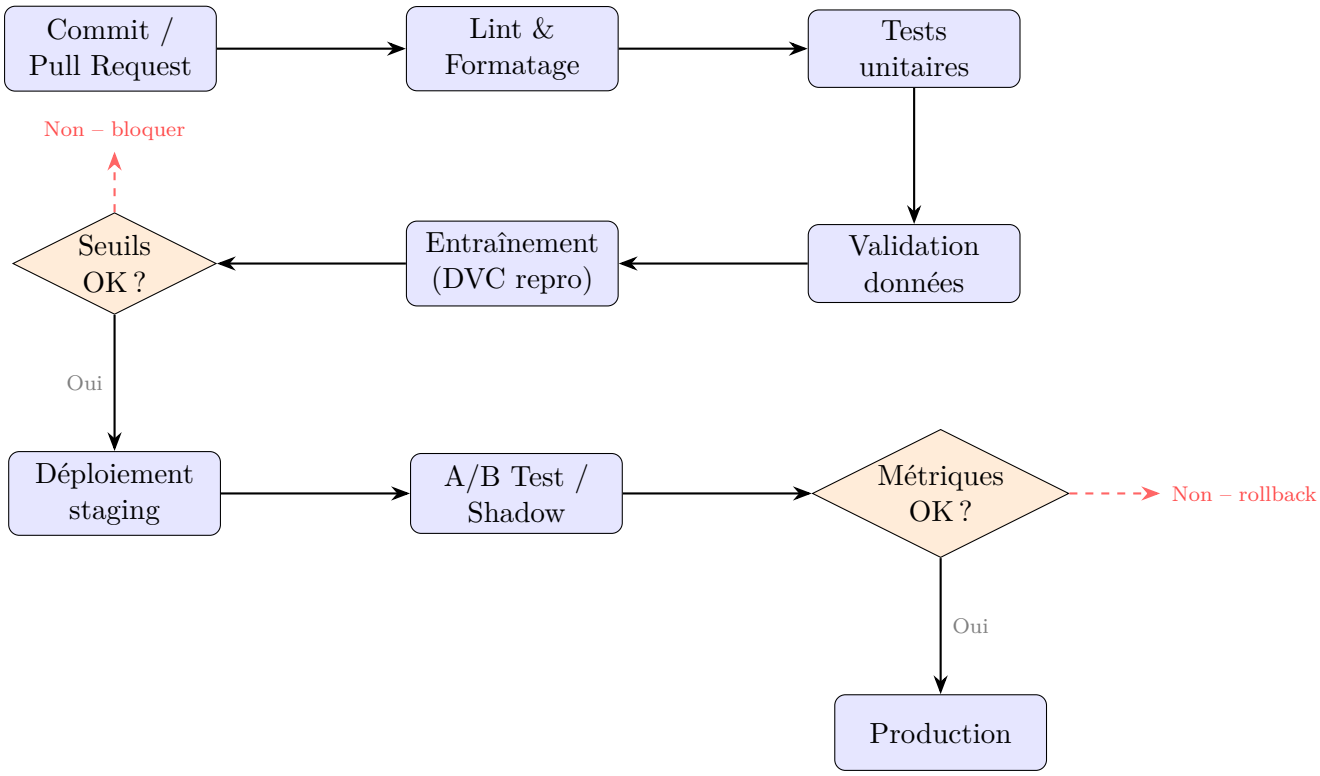


FIG. 9.1 : Pipeline CI/CD complet pour un projet de Machine Learning.

9.7 Architecture d'un pipeline CI/CD pour le ML

9.8 Comparaison des outils CI/CD

GitHub Actions vs GitLab CI vs Jenkins vs CircleCI

Critère	GitHub Actions	GitLab CI	Jenkins	CircleCI
Hébergement	Cloud	Cloud/Self	Self-hosted	Cloud
Configuration	YAML	YAML	Groovy/YAML	YAML
GPU natif	Non*	Oui	Oui	Non*
Cache intégré	Oui	Oui	Plugin	Oui
Marketplace	Très riche	Modéré	Très riche	Modéré
Gratuit (public)	Oui	Oui	Oui	Limité
Complexité	Faible	Faible	Élevée	Faible
Intégration ML	Bonne	Bonne	Manuelle	Modérée

* Possible via des runners auto-hébergés.

Erreurs fréquentes en CI/CD ML

- **Stocker les données dans Git** : les fichiers volumineux ralentissent le dépôt et augmentent les coûts. Utilisez DVC ou Git LFS.
- **Entraîner sur la CI sans cache** : chaque pipeline ré-entraîne depuis zéro.

Utilisez le cache DVC.

- **Ignorer les tests de données** : tester uniquement le code ne suffit pas en ML.
- **Déployer sans *gate*** : l'absence de seuils de performance peut mettre un modèle dégradé en production.
- **Secrets en dur** : ne jamais mettre les clés API dans le code. Utilisez les *secrets* du fournisseur CI.

9.9 Mini-projet : mettre en place le CI/CD

CI/CD pour le projet de cours

Objectif : configurer un pipeline CI/CD complet pour le projet ML du cours.
Étapes :

1. **Pre-commit** : créer le fichier `.pre-commit-config.yaml` avec Black, Ruff et mypy. Exécuter `pre-commit run --all-files` et corriger les erreurs.
2. **Tests** : écrire au moins 5 tests unitaires (fonctions de prétraitement, chargement des données, forme des sorties du modèle).
3. **Validation** : créer le script `scripts/validate_model.py` avec des seuils de performance.
4. **GitHub Actions** : créer `.github/workflows/ml-pipeline.yml` avec les jobs : lint, test, train, validate.
5. **DVC** : intégrer `dvc repro` dans le pipeline CI.
6. **Badge** : ajouter le badge de statut CI dans le `README.md`.

Livrables :

- Dépôt GitHub avec pipeline CI fonctionnel (badge vert).
- Capture d'écran de l'exécution réussie du workflow.
- Document expliquant chaque étape du pipeline.

9.10 Exercices

Exercice 9.1 (Niveau 1 – Premiers pas avec GitHub Actions)**.** Créez un workflow GitHub Actions minimal qui :

1. Se déclenche sur chaque push vers la branche `main`.
2. Installe Python 3.11 et les dépendances du projet.

3. Exécute `ruff check src/` et `pytest tests/`.

Vérifiez que le workflow s'exécute correctement en poussant un commit.

Exercice 9.2 (Niveau 2 – Validation automatisée du modèle)**.** Étendez le pipeline CI pour inclure une étape de validation du modèle :

1. Après l'entraînement, exécutez un script qui vérifie que l'accuracy dépasse 0.80 et que le F1-score dépasse 0.75.
2. Ajoutez une vérification de drift via le test de Kolmogorov-Smirnov entre les données de référence et les données courantes.
3. Si un seuil n'est pas atteint, le pipeline doit échouer avec un message explicite.
4. Utilisez `dvc metrics diff` pour comparer les métriques avec la branche principale et afficher la différence dans un commentaire de PR.

Exercice 9.3 (Niveau 3 – Pipeline CI/CD de bout en bout avec A/B testing)**.** Concevez et implémentez un pipeline CI/CD complet :

1. **CI** : lint, tests, entraînement, validation (seuils + drift + latence).
2. **CD staging** : déploiement automatique sur un environnement de staging via Docker + Kubernetes.
3. **A/B testing** : configurez Istio ou un proxy pour router 10 % du trafic vers le nouveau modèle pendant 24h.
4. **Promotion** : si les métriques métier (taux de conversion, latence p95) sont meilleures, promouvoir automatiquement le modèle en production.
5. **Rollback** : implémentez un mécanisme de rollback automatique si les métriques se dégradent.

Documentez l'architecture complète avec un diagramme et justifiez vos choix techniques.

9.11 Aide-mémoire

CI/CD pour le Machine Learning

Concepts clés :

- **CI** = intégration continue : tests automatiques à chaque commit
- **CD** = livraison/déploiement continu : mise en production automatisée
- **Gate** = seuil de qualité à franchir avant déploiement

Commandes essentielles :

```
# Pre-commit
pre-commit install          # Activer les hooks
```

```
pre-commit run --all-files  # Executer sur tout

# DVC dans la CI
dvc pull                    # Telecharger les donnees
dvc repro                   # Reproduire le pipeline
dvc metrics diff main       # Comparer les metriques

# GitHub Actions (debug local)
act -j test                 # Executer un job localement
```

Checklist CI/CD ML :

- ☐ Pre-commit hooks configurés (Black, Ruff, mypy)
- ☐ Tests unitaires avec couverture $\geq 80\,\%$
- ☐ Validation des données (schéma, drift)
- ☐ Seuils de performance définis et vérifiés
- ☐ Vérification de la latence d'inférence
- ☐ Pipeline DVC reproductible
- ☐ Secrets stockés de manière sécurisée
- ☐ Stratégie de rollback définie

Chapitre 10

Monitoring des Modèles en Production

10.1 Introduction

Déployer un modèle de Machine Learning en production n'est pas la fin du processus : c'est le début d'une nouvelle phase critique. Les modèles se dégradent inévitablement avec le temps, car les données du monde réel évoluent. Le monitoring permet de détecter ces dégradations avant qu'elles n'impactent les utilisateurs.

Pourquoi les modèles se dégradent-ils ?

Un modèle entraîné capture une relation $f : X \to Y$ valide au moment de l'entraînement. Cette relation peut devenir obsolète pour plusieurs raisons :

- **Les données changent** : le profil des utilisateurs évolue, de nouveaux comportements apparaissent.
- **Le contexte change** : saisonnalité, événements économiques, changements réglementaires.
- **Le système en amont change** : modification d'un capteur, changement de format de données.
- **Les labels changent** : ce qui était considéré comme positif hier ne l'est plus aujourd'hui.

Sans monitoring, ces dégradations passent inaperçues pendant des semaines ou des mois.

10.2 Types de drift

Le *drift* (dérive) désigne tout changement dans la distribution des données ou dans la relation entre les entrées et les sorties d'un modèle.

Définition 10.1 (Covariate Shift)**.** Le *covariate shift* (dérive des covariables) survient lorsque la distribution des variables d'entrée $P(X)$ change, tandis que la relation conditionnelle $P(Y|X)$ reste identique. Par exemple, un modèle de crédit entraîné sur des clients

de 25–55 ans qui reçoit soudainement des demandes de clients de 18–25 ans.

$$P_{\text{train}}(X) \neq P_{\text{prod}}(X), \quad P_{\text{train}}(Y|X) = P_{\text{prod}}(Y|X)$$

Définition 10.2 (Prior Probability Shift)**.** Le *prior probability shift* (dérive de la distribution cible) survient lorsque la distribution de la variable cible $P(Y)$ change. Par exemple, un classificateur de spam entraîné avec 10 % de spam qui fait face à 40 % de spam en production.

$$P_{\text{train}}(Y) \neq P_{\text{prod}}(Y), \quad P_{\text{train}}(X|Y) = P_{\text{prod}}(X|Y)$$

Définition 10.3 (Concept Drift)**.** Le *concept drift* (dérive de concept) est le cas le plus sévère : la relation entre les entrées et la cible change fondamentalement.

$$P_{\text{train}}(Y|X) \neq P_{\text{prod}}(Y|X)$$

Exemple : un modèle de recommandation dont le comportement des utilisateurs change radicalement après une crise économique.

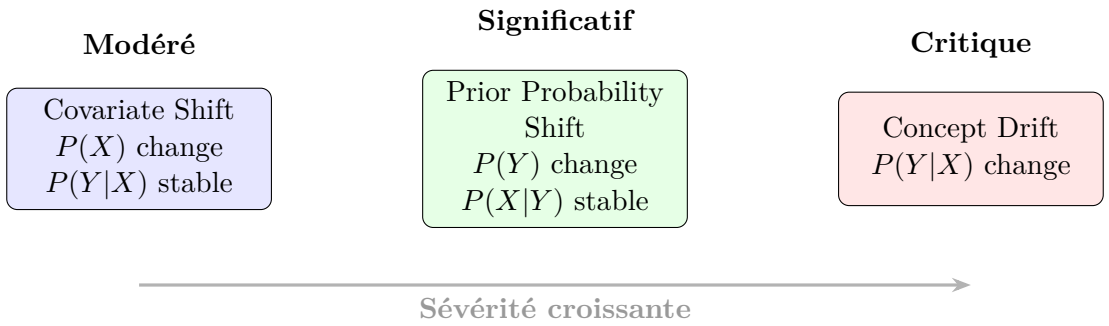


FIG. 10.1 : Les trois types de drift classés par sévérité.

10.3 Formulation mathématique de la détection de drift

Métriques de détection de drift

Divergence de Kullback-Leibler (KL) :

$$D_{\text{KL}}(P\|Q) = \sum_{x} P(x) \ln \frac{P(x)}{Q(x)}$$

La divergence KL est **asymétrique** : $D_{\text{KL}}(P\|Q) \neq D_{\text{KL}}(Q\|P)$. Elle mesure la quantité d'information perdue lorsqu'on utilise Q pour approximer P.

Population Stability Index (PSI) :

$$\text{PSI} = \sum_{i=1}^{k} (p_i - q_i) \ln \frac{p_i}{q_i}$$

où p_i et q_i sont les proportions dans le i-ème intervalle (*bin*) des distributions de référence et courante. Règles d'interprétation :

- PSI < 0.1 : pas de changement significatif
- $0.1 \leq$ PSI < 0.2 : changement modéré, à surveiller
- PSI ≥ 0.2 : changement significatif, action requise

Test de Kolmogorov-Smirnov (KS) :

$$D_n = \sup_x |F_{\text{ref}}(x) - F_{\text{cur}}(x)|$$

où F_{ref} et F_{cur} sont les fonctions de répartition empiriques des distributions de référence et courante. On rejette l'hypothèse nulle (« pas de drift ») si la *p-value* est inférieure au seuil α (typiquement 0.05).

Implémentation des métriques de drift

```python
import numpy as np
from scipy import stats


def kl_divergence(p: np.ndarray, q: np.ndarray,
                  epsilon: float = 1e-10) -> float:
    """Divergence KL entre deux distributions discretes."""
    p = np.clip(p, epsilon, None)
    q = np.clip(q, epsilon, None)
    p = p / p.sum()
    q = q / q.sum()
    return float(np.sum(p * np.log(p / q)))


def psi(reference: np.ndarray, current: np.ndarray,
        n_bins: int = 10) -> float:
    """Population Stability Index."""
    breakpoints = np.linspace(
        min(reference.min(), current.min()),
        max(reference.max(), current.max()),
        n_bins + 1,
    )
    ref_counts = np.histogram(reference, bins=breakpoints)[0]
    cur_counts = np.histogram(current, bins=breakpoints)[0]

    # Eviter les zeros
    ref_pct = (ref_counts + 1) / (len(reference) + n_bins)
    cur_pct = (cur_counts + 1) / (len(current) + n_bins)

    return float(np.sum((cur_pct - ref_pct) * np.log(cur_pct / ref_pct)))


def ks_test(reference: np.ndarray, current: np.ndarray,
            alpha: float = 0.05) -> dict:
```

```
    """Test de Kolmogorov-Smirnov pour detecter le drift."""
    statistic, p_value = stats.ks_2samp(reference, current)
    return {
        "statistic": float(statistic),
        "p_value": float(p_value),
        "drift_detected": p_value < alpha,
    }
```

10.4 Evidently AI : rapports de monitoring

Evidently est une bibliothèque open-source Python qui génère des rapports de monitoring pour les modèles ML. Elle permet de détecter le drift, d'analyser la qualité des données et de suivre les performances du modèle.

Générer un rapport de drift avec Evidently

```
import pandas as pd
from evidently import ColumnMapping
from evidently.report import Report
from evidently.metric_preset import (
    DataDriftPreset,
    DataQualityPreset,
    TargetDriftPreset,
)


# Charger les donnees
reference_data = pd.read_csv("data/reference.csv")
current_data = pd.read_csv("data/production_week42.csv")

# Definir le mapping des colonnes
column_mapping = ColumnMapping(
    target="label",
    prediction="prediction",
    numerical_features=["age", "income", "credit_score"],
    categorical_features=["region", "product_type"],
)

# Creer le rapport de drift
drift_report = Report(metrics=[
    DataDriftPreset(),
    DataQualityPreset(),
    TargetDriftPreset(),
])

drift_report.run(
    reference_data=reference_data,
    current_data=current_data,
```

```
    column_mapping=column_mapping,
)

# Sauvegarder en HTML
drift_report.save_html("reports/drift_report_week42.html")
```

Tests automatisés avec Evidently

```
from evidently.test_suite import TestSuite
from evidently.test_preset import (
    DataDriftTestPreset,
    DataStabilityTestPreset,
    NoTargetPerformanceTestPreset,
)
from evidently.tests import (
    TestColumnDrift,
    TestShareOfDriftedColumns,
    TestMeanInNSigmas,
)


# Suite de tests pour la CI
test_suite = TestSuite(tests=[
    # Preset : tester le drift sur toutes les colonnes
    DataDriftTestPreset(),

    # Tests specifiques
    TestColumnDrift(column_name="income", stattest="ks"),
    TestShareOfDriftedColumns(lt=0.3),  # < 30% de drift
    TestMeanInNSigmas(column_name="age", n=2),
])

test_suite.run(
    reference_data=reference_data,
    current_data=current_data,
    column_mapping=column_mapping,
)

# Verifier si les tests passent
result = test_suite.as_dict()
all_passed = all(
    t["status"] == "SUCCESS"
    for t in result["tests"]
)

if not all_passed:
    failed = [
        t["name"] for t in result["tests"]
        if t["status"] == "FAIL"
    ]
```

```
    raise RuntimeError(
        f"Tests de drift echoues : {failed}"
    )

print("Tous les tests de monitoring sont passes.")
```

10.5 Prometheus et Grafana pour la collecte de métriques

Prometheus est un système de monitoring et d'alerte open-source qui collecte des métriques sous forme de séries temporelles. Grafana est un outil de visualisation qui permet de créer des tableaux de bord à partir de ces métriques.

Exposer des métriques ML avec Prometheus

```
from prometheus_client import (
    Counter, Histogram, Gauge, Summary,
    start_http_server,
)
import time
import numpy as np

# Definir les metriques
PREDICTION_COUNT = Counter(
    "model_predictions_total",
    "Nombre total de predictions",
    ["model_version", "outcome"],
)

PREDICTION_LATENCY = Histogram(
    "model_prediction_latency_seconds",
    "Latence des predictions en secondes",
    buckets=[0.01, 0.025, 0.05, 0.1, 0.25, 0.5, 1.0],
)

FEATURE_DRIFT_SCORE = Gauge(
    "model_feature_drift_psi",
    "Score PSI de drift par feature",
    ["feature_name"],
)

PREDICTION_CONFIDENCE = Summary(
    "model_prediction_confidence",
    "Distribution de la confiance des predictions",
)


class MonitoredModel:
```

```python
    """Wrapper qui ajoute le monitoring a un modele."""

    def __init__(self, model, version: str = "v1"):
        self.model = model
        self.version = version

    def predict(self, features: np.ndarray) -> np.ndarray:
        start = time.perf_counter()

        predictions = self.model.predict(features)
        probabilities = self.model.predict_proba(features)

        # Enregistrer la latence
        latency = time.perf_counter() - start
        PREDICTION_LATENCY.observe(latency)

        # Enregistrer les predictions
        for pred in predictions:
            PREDICTION_COUNT.labels(
                model_version=self.version,
                outcome=str(pred),
            ).inc()

        # Enregistrer la confiance
        for prob in probabilities.max(axis=1):
            PREDICTION_CONFIDENCE.observe(prob)

        return predictions

    def update_drift_metrics(
        self, feature_name: str, psi_value: float
    ):
        """Met a jour le score de drift pour une feature."""
        FEATURE_DRIFT_SCORE.labels(
            feature_name=feature_name
        ).set(psi_value)


# Demarrer le serveur de metriques sur le port 8000
start_http_server(8000)
```

Configuration Prometheus `prometheus.yml`

```yaml
global:
  scrape_interval: 15s
  evaluation_interval: 15s

rule_files:
  - "ml_alerts.yml"
```

```yaml
scrape_configs:
  - job_name: "ml-model"
    static_configs:
      - targets: ["ml-model-service:8000"]
    metrics_path: /metrics
    scrape_interval: 10s

  - job_name: "ml-drift-checker"
    static_configs:
      - targets: ["drift-service:8001"]
    scrape_interval: 300s  # Toutes les 5 minutes
```

10.6 Alertes et tableaux de bord

Règles d'alerte Prometheus `ml_alerts.yml`

```yaml
groups:
  - name: ml-model-alerts
    rules:
      - alert: HighPredictionLatency
        expr: |
          histogram_quantile(0.95,
            rate(model_prediction_latency_seconds_bucket[5m])
          ) > 0.5
        for: 10m
        labels:
          severity: warning
        annotations:
          summary: "Latence p95 > 500ms depuis 10min"
          description: >
            La latence au 95e percentile est de
            {{ $value | humanizeDuration }}.

      - alert: DataDriftDetected
        expr: model_feature_drift_psi > 0.2
        for: 30m
        labels:
          severity: critical
        annotations:
          summary: "Drift detecte sur {{ $labels.feature_name }}"
          description: >
            PSI = {{ $value }} (seuil: 0.2).
            Re-entrainement recommande.

      - alert: LowPredictionVolume
        expr: |
          rate(model_predictions_total[1h]) < 10
        for: 30m
```

```
        labels:
          severity: warning
        annotations:
          summary: "Volume de predictions anormalement bas"

      - alert: LowModelConfidence
        expr: |
          model_prediction_confidence{quantile="0.5"} < 0.6
        for: 15m
        labels:
          severity: warning
        annotations:
          summary: "Confiance mediane < 60%"
```

Tableau de bord Grafana – métriques essentielles

Un bon tableau de bord ML doit inclure les panneaux suivants :

1. **Volume de prédictions** : nombre de requêtes par minute (détecter les anomalies de trafic).
2. **Distribution des prédictions** : répartition des classes prédites (détecter le *prior probability shift*).
3. **Latence** : p50, p95, p99 de la latence d'inférence.
4. **Confiance** : distribution des scores de confiance du modèle.
5. **Drift par feature** : score PSI pour chaque variable d'entrée.
6. **Performance** : accuracy, F1-score glissant (si les labels sont disponibles).
7. **Erreurs système** : taux d'erreur HTTP, utilisation CPU/mémoire.

10.7 Architecture de monitoring

10.8 Comparaison des outils de monitoring ML

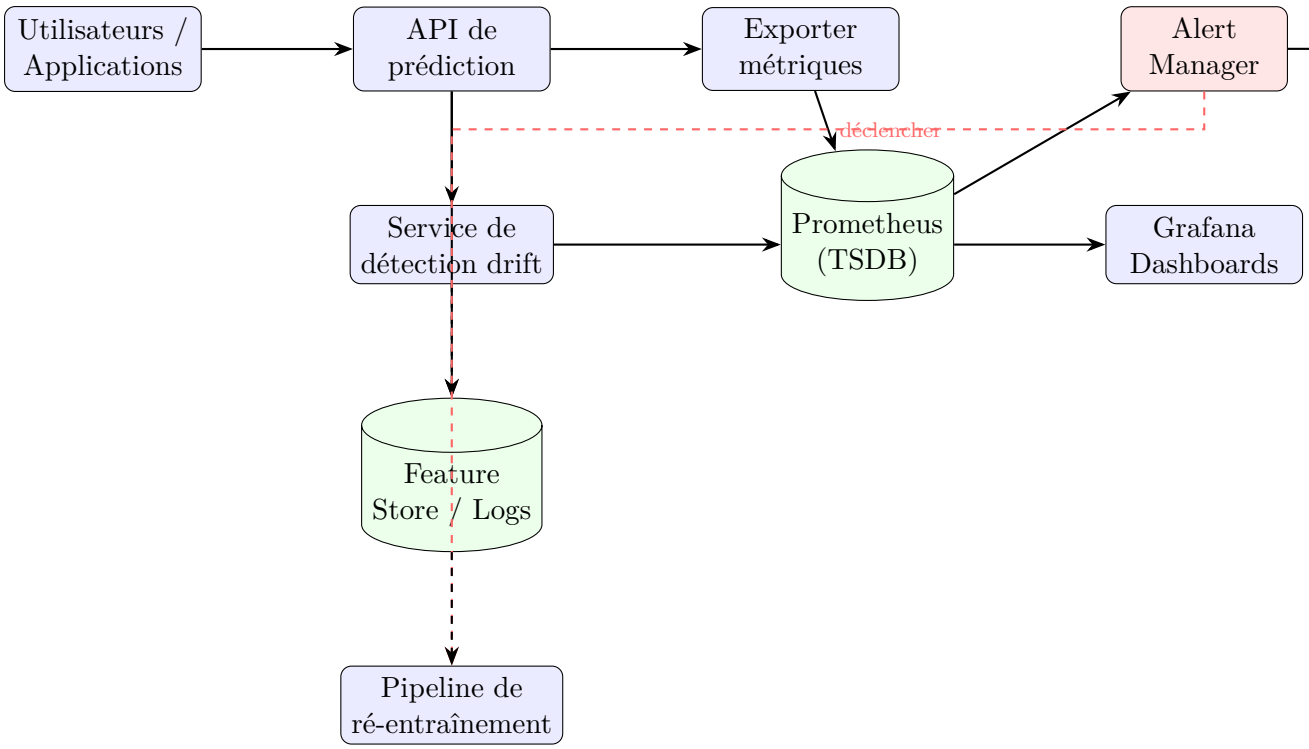


FIG. 10.2 : Architecture de monitoring pour un modèle ML en production.

Evidently vs WhyLabs vs Fiddler vs NannyML

Critère	Evidently	WhyLabs	Fiddler	NannyML
Licence	Open-source	Freemium	Commercial	Open-source
Détection drift	Oui	Oui	Oui	Oui
Qualité données	Oui	Oui	Limitée	Non
Explicabilité	Limitée	Non	Oui	Non
Perf. sans labels	Non	Oui	Oui	Oui
Intégration CI	Oui	Oui	Non	Oui
Dashboard	HTML/Cloud	Cloud	Cloud	Limité
Complexité	Faible	Modérée	Élevée	Faible

Remarque 10.4. NannyML se distingue par sa capacité à estimer la performance d'un modèle *sans avoir accès aux labels réels*, grâce à la méthode CBPE (*Confidence-Based Performance Estimation*). Cela est particulièrement utile lorsque les labels arrivent avec un délai important (ex. : détection de fraude, prédiction de churn à 90 jours).

Stack de monitoring recommandée

Pour un projet MLOps complet, la stack suivante est recommandée :

- **Evidently** pour la détection de drift et les tests en CI
- **Prometheus** pour la collecte de métriques en temps réel
- **Grafana** pour la visualisation et les tableaux de bord

- **AlertManager** pour les notifications (Slack, PagerDuty, email)
- **NannyML** pour l'estimation de performance sans labels

Cette combinaison offre une couverture complète tout en restant open-source.

10.9 Mini-projet : ajouter le monitoring au modèle déployé

Monitoring du modèle en production

Objectif : ajouter une couche de monitoring complète au modèle déployé lors des chapitres précédents.
Étapes :

1. **Instrumenter l'API** : ajouter les métriques Prometheus (compteur de prédictions, histogramme de latence, jauge de drift) à l'API FastAPI/Flask.
2. **Logger les prédictions** : stocker chaque prédiction avec ses features d'entrée dans un fichier CSV ou une base de données.
3. **Script de drift** : créer un script qui compare les données de référence aux données de production de la dernière semaine (PSI + KS test).
4. **Rapport Evidently** : générer un rapport HTML de drift hebdomadaire automatisé via un cron job ou un workflow GitHub Actions.
5. **Prometheus + Grafana** : déployer Prometheus et Grafana via Docker Compose. Créer un dashboard avec au moins 5 panneaux.
6. **Alertes** : configurer au moins 3 alertes (latence élevée, drift détecté, volume anormal).

Docker Compose :

```yaml
version: "3.8"
services:
  ml-model:
    build: .
    ports:
      - "8080:8080"
      - "8000:8000"  # Metriques Prometheus

  prometheus:
    image: prom/prometheus:latest
    volumes:
      - ./monitoring/prometheus.yml:/etc/prometheus/prometheus.yml
      - ./monitoring/ml_alerts.yml:/etc/prometheus/ml_alerts.yml
    ports:
```

```
      - "9090:9090"

  grafana:
    image: grafana/grafana:latest
    ports:
      - "3000:3000"
    environment:
      - GF_SECURITY_ADMIN_PASSWORD=admin
    volumes:
      - grafana-data:/var/lib/grafana

volumes:
  grafana-data:
```

Livrables :

- Code source de l'API instrumentée avec les métriques Prometheus.
- Fichier `docker-compose.yml` pour le déploiement complet.
- Capture d'écran du dashboard Grafana avec les métriques ML.
- Rapport Evidently généré automatiquement.
- Documentation des alertes configurées.

10.10 Exercices

Exercice 10.1 (Niveau 1 – Détection de drift basique).

1. Générez deux distributions normales : $X_{\text{ref}} \sim \mathcal{N}(0, 1)$ avec 10 000 échantillons et $X_{\text{cur}} \sim \mathcal{N}(0.5, 1.2)$ avec 10 000 échantillons.
2. Calculez le PSI entre les deux distributions. Interprétez le résultat.
3. Appliquez le test de Kolmogorov-Smirnov. Le drift est-il détecté avec $\alpha = 0.05$?
4. Calculez la divergence KL dans les deux sens. Observez l'asymétrie.
5. Utilisez Evidently pour générer un rapport de drift sur ces données synthétiques.

Exercice 10.2 (Niveau 2 – Monitoring d'une API de prédiction).

1. Créez une API FastAPI qui sert un modèle scikit-learn (classificateur de votre choix).
2. Ajoutez les métriques Prometheus suivantes : nombre de prédictions, latence (histogramme), distribution des classes prédites.
3. Déployez Prometheus et Grafana avec Docker Compose.
4. Créez un dashboard Grafana affichant les métriques en temps réel.
5. Écrivez un script de charge qui envoie des requêtes avec des données progressivement driftées.

6. Configurez une alerte qui se déclenche lorsque la latence p95 dépasse 200 ms.

Exercice 10.3 (Niveau 3 – Système de monitoring autonome avec re-entraînement)**.** Concevez et implémentez un système de monitoring complet :

1. **Détection multi-méthode** : implémentez PSI, KS, et la méthode ADWIN (*Adaptive Windowing*) pour la détection de drift en temps réel.
2. **Pipeline de monitoring** : créez un service qui :
   - Collecte les prédictions et features toutes les 5 minutes
   - Calcule les scores de drift sur une fenêtre glissante
   - Met à jour les métriques Prometheus
3. **Re-entraînement automatique** : lorsqu'un drift significatif est détecté (PSI $>$ 0.2 pendant plus de 30 minutes), déclenchez automatiquement un pipeline de re-entraînement via l'API GitHub Actions.
4. **Validation post-entraînement** : le nouveau modèle doit passer les tests de performance avant déploiement (*champion/challenger*).
5. **Observabilité complète** : dashboard Grafana avec au moins 10 panneaux couvrant performance, drift, infrastructure et métriques métier.

10.11 Aide-mémoire

Monitoring des modèles ML

Types de drift :

- **Covariate shift** : $P(X)$ change, $P(Y|X)$ stable
- **Prior probability shift** : $P(Y)$ change
- **Concept drift** : $P(Y|X)$ change (le plus grave)

Métriques de drift :

- **PSI** : < 0.1 OK, 0.1–0.2 surveiller, ≥ 0.2 agir
- **KS test** : $p < 0.05 \Rightarrow$ drift détecté
- **KL divergence** : asymétrique, ≥ 0, $= 0$ si identiques

Code Evidently essentiel :

```
from evidently.report import Report
from evidently.metric_preset import DataDriftPreset

report = Report(metrics=[DataDriftPreset()])
report.run(
    reference_data=ref_df,
```

```
    current_data=cur_df,
)
report.save_html("drift_report.html")
```

Prometheus – métriques clés :

```
from prometheus_client import Counter, Histogram, Gauge
PREDICTIONS = Counter("predictions_total", "...", ["class"])
LATENCY = Histogram("prediction_latency_s", "...")
DRIFT = Gauge("feature_drift_psi", "...", ["feature"])
```

Checklist monitoring :

- ☐ Métriques Prometheus exposées par l'API
- ☐ Prédictions et features loguées
- ☐ Détection de drift automatisée
- ☐ Dashboard Grafana opérationnel
- ☐ Alertes configurées (latence, drift, volume)
- ☐ Processus de re-entraînement défini
- ☐ Stratégie de rollback prête

Chapitre 11

Reproductibilité en Recherche — Standards et Bonnes Pratiques

11.1 La crise de la reproductibilité en ML

La **crise de la reproductibilité** est un problème majeur en recherche scientifique, et le machine learning n'y échappe pas. Plusieurs études ont montré que la majorité des résultats publiés en ML sont difficiles, voire impossibles, à reproduire.

Chiffres alarmants

- Selon une enquête de *Nature* (2016), plus de 70 % des chercheurs ont échoué à reproduire les résultats d'un autre chercheur.
- En NLP, Dodge et al. (2019) ont montré que le simple choix de la graine aléatoire pouvait faire varier la performance de 1 à 5 points de F1-score.
- En vision par ordinateur, de nombreux résultats « état de l'art » ne sont pas reproductibles sans accès au code et aux données originaux.

Les causes spécifiques au ML sont multiples :

- **Aléatoire non contrôlé** : initialisation des poids, ordre des batchs, augmentation de données.
- **Dépendances logicielles** : versions de CUDA, cuDNN, PyTorch qui influencent les résultats numériques.
- **Données non partagées** : datasets propriétaires, prétraitements non documentés.
- **Hyperparammètres cachés** : paramètres non mentionnés dans l'article (learning rate scheduler, warmup, etc.).
- **Coût computationnel** : entraînements nécessitant des centaines de GPU-heures, impossibles à re-exécuter.

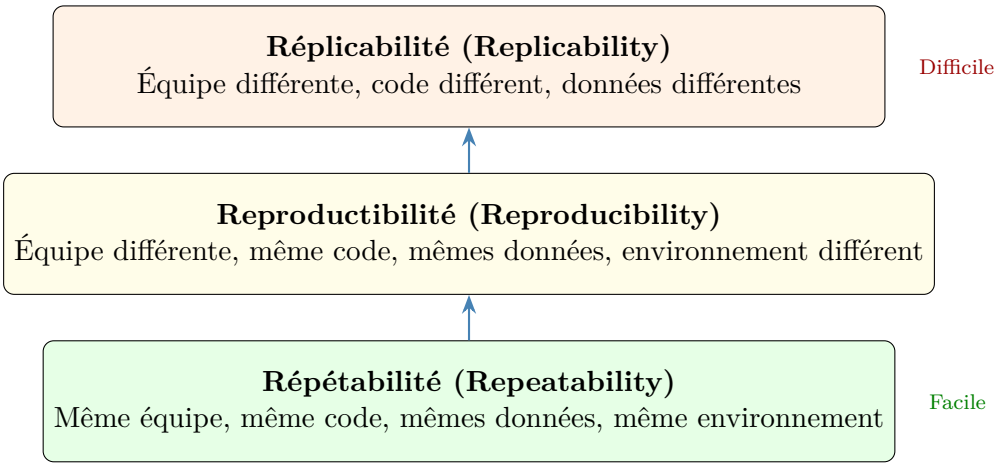
11.2 Niveaux de reproductibilité

Il est essentiel de distinguer trois niveaux de reproductibilité, souvent confondus :

Définition 11.1 (Répétabilité)**.** La **répétabilité** (*repeatability*) est la capacité d'obtenir les mêmes résultats en ré-exécutant le même code, sur les mêmes données, dans le même environnement. C'est le niveau le plus basique.

Définition 11.2 (Reproductibilité)**.** La **reproductibilité** (*reproducibility*) est la capacité pour une équipe indépendante d'obtenir les mêmes résultats en utilisant le code et les données fournis par les auteurs, mais dans leur propre environnement.

Définition 11.3 (Réplicabilité)**.** La **réplicabilité** (*replicability*) est la capacité d'obtenir des résultats similaires avec une implémentation indépendante, sur des données différentes. C'est le niveau le plus exigeant, qui valide la généralisabilité de la méthode.



Remarque 11.4. En pratique, même la répétabilité n'est pas garantie en ML : les opérations GPU non déterministes (atomicAdd, cuDNN autotuner) peuvent produire des résultats légèrement différents d'une exécution à l'autre sur le même matériel.

11.3 Gestion des graines aléatoires

Le contrôle de l'aléatoire est la première étape vers la répétabilité. En Python, plusieurs sources d'aléatoire coexistent et doivent toutes être fixées.

Configuration complète de reproductibilité

```python
"""Reproducibility utilities for PyTorch projects."""
import os
import random
import numpy as np
import torch


def set_seed(seed: int = 42, deterministic: bool = True) -> None:
    """Set all random seeds for reproducibility.

    Args:
        seed: Random seed value.
        deterministic: If True, enforce deterministic algorithms
            (may reduce performance).
```

```
    """
    # Python built-in random
    random.seed(seed)

    # Numpy
    np.random.seed(seed)

    # PyTorch CPU
    torch.manual_seed(seed)

    # PyTorch GPU (all devices)
    torch.cuda.manual_seed(seed)
    torch.cuda.manual_seed_all(seed)

    # Hash seed for Python dicts, sets, etc.
    os.environ["PYTHONHASHSEED"] = str(seed)

    if deterministic:
        # Force deterministic algorithms
        torch.use_deterministic_algorithms(True)

        # Disable cuDNN benchmark (non-deterministic)
        torch.backends.cudnn.benchmark = False
        torch.backends.cudnn.deterministic = True

        # Handle CUBLAS non-determinism
        os.environ["CUBLAS_WORKSPACE_CONFIG"] = ":4096:8"


def seed_worker(worker_id: int) -> None:
    """Seed function for DataLoader workers.

    Usage:
        DataLoader(..., worker_init_fn=seed_worker)
    """
    worker_seed = torch.initial_seed() % 2**32
    np.random.seed(worker_seed)
    random.seed(worker_seed)


# Usage
set_seed(42)

# DataLoader with reproducible workers
g = torch.Generator()
g.manual_seed(42)

train_loader = torch.utils.data.DataLoader(
    dataset,
    batch_size=32,
    shuffle=True,
```

```
    num_workers=4,
    worker_init_fn=seed_worker,
    generator=g,
)
```

Déterminisme CUDA : le prix à payer

Activer `torch.use_deterministic_algorithms(True)` peut :

- Ralentir l'entraînement de 10 à 20 %.
- Lever des erreurs pour certaines opérations qui n'ont pas d'implémentation déterministe (ex. : `scatter_add`, certaines convolutions).
- Ne pas être suffisant entre versions de CUDA ou de matériel différents.

Recommandation : utiliser le déterminisme pour le débogage et la validation, mais accepter une légère variabilité en production.

Sources d'aléatoire à contrôler

Source	Commande	Impact
Python `random`	`random.seed(s)`	Shuffling, sampling
NumPy	`np.random.seed(s)`	Initialisation, bruit
PyTorch CPU	`torch.manual_seed(s)`	Poids initiaux
PyTorch GPU	`torch.cuda.manual_seed_all(s)`	Opérations CUDA
cuDNN	`cudnn.deterministic = True`	Convolutions
CUBLAS	`CUBLAS_WORKSPACE_CONFIG`	Produits matriciels
DataLoader	`worker_init_fn`	Workers parallèles
PYTHONHASHSEED	`os.environ[...] = str(s)`	Ordre des dicts

11.4 Documentation des expériences

Au-delà du code, la documentation est essentielle pour la reproductibilité. Deux standards émergent dans la communauté ML.

11.4.1 Model Cards

Les **Model Cards** (Mitchell et al., 2019) sont des documents structurés qui accompagnent un modèle publié.

Contenu d'une Model Card

1. **Détails du modèle** : architecture, version, auteurs, licence.
2. **Usage prévu** : cas d'utilisation primaires et hors-scope.
3. **Métriques** : métriques choisies et justification.

4. **Données d'entraînement** : description, taille, source.
5. **Données d'évaluation** : dataset de test, méthodologie.
6. **Analyses éthiques** : biais, équité, limites connues.
7. **Résultats quantitatifs** : par sous-groupe démographique si applicable.
8. **Considérations environnementales** : empreinte carbone, coût computationnel.

Model Card en YAML (extrait)

```yaml
model_card:
  model_details:
    name: "sentiment-classifier-v2"
    version: "2.1.0"
    architecture: "BERT-base fine-tuned"
    parameters: 110_000_000
    license: "Apache-2.0"
    training_date: "2025-01-15"
    training_cost: "8 GPU-hours (A100)"

  intended_use:
    primary: "Sentiment analysis of product reviews"
    out_of_scope: "Medical text, legal documents"

  metrics:
    - name: "F1-score (macro)"
      value: 0.892
      confidence_interval: [0.885, 0.899]
    - name: "Accuracy"
      value: 0.904

  training_data:
    name: "Amazon Reviews (2023)"
    size: "3.6M reviews"
    preprocessing: "Lowercased, truncated to 512 tokens"

  ethical_considerations:
    bias: "Lower performance on non-English reviews"
    fairness_evaluation: "Tested across 5 product categories"
```

11.4.2 Datasheets for Datasets

Les **Datasheets for Datasets** (Gebru et al., 2021) documentent les datasets de manière standardisée :

- **Motivation** : pourquoi le dataset a-t-il été créé ?
- **Composition** : quelles instances contient-il ? Quelle est la distribution ?

- **Collecte** : comment les données ont-elles été collectées ? Par qui ?
- **Prétraitement** : quelles transformations ont été appliquées ?
- **Distribution** : comment le dataset est-il distribué ? Sous quelle licence ?
- **Maintenance** : qui maintient le dataset ? Comment signaler des problèmes ?

11.5 Archivage et partage des artefacts

La reproductibilité exige que le code, les données et les modèles soient archivés de manière pérenne et accessible.

Plateformes d'archivage

Plateforme	Type	Cas d'usage
Zenodo	Code + données	Archive pérenne avec DOI, intégration GitHub, jusqu'à 50 Go par dépôt
Papers with Code	Code + résultats	Lier articles, code et benchmarks ; leaderboards communautaires
Hugging Face Hub	Modèles + datasets	Hébergement de modèles et datasets avec versionnement
GitHub Releases	Code	Snapshots versionnés du code
DVC (remote)	Données + modèles	Versionnement de fichiers volumineux sur S3, GCS, Azure

Archivage automatique avec Zenodo + GitHub

```
# 1. Activer l'integration Zenodo-GitHub
#    -> https://zenodo.org/account/settings/github/

# 2. Creer une release GitHub
git tag -a v1.0.0 -m "Camera-ready version for NeurIPS 2025"
git push origin v1.0.0

# 3. Zenodo cree automatiquement un DOI pour la release

# 4. Ajouter le badge DOI dans le README
# [![DOI](https://zenodo.org/badge/DOI/10.5281/zenodo.XXXXX.svg)]
```

Publication sur Hugging Face Hub

```
from huggingface_hub import HfApi, ModelCard

api = HfApi()

# Upload model
```

```
api.upload_folder(
    folder_path="./model_checkpoint",
    repo_id="username/sentiment-classifier-v2",
    repo_type="model",
)

# Upload dataset
api.upload_folder(
    folder_path="./processed_data",
    repo_id="username/amazon-reviews-2023",
    repo_type="dataset",
)

# Create and push model card
card = ModelCard.load("username/sentiment-classifier-v2")
card.data.language = "en"
card.data.license = "apache-2.0"
card.data.datasets = ["username/amazon-reviews-2023"]
card.push_to_hub("username/sentiment-classifier-v2")
```

11.6 Checklist de reproductibilité

La conférence NeurIPS exige depuis 2019 une checklist de reproductibilité. Voici une version adaptée pour tout projet ML :

Checklist de reproductibilité ML

Code et environnement :

- ☐ Code source disponible publiquement (ou en supplément)
- ☐ Fichier de dépendances complet avec versions épinglées
- ☐ Instructions d'installation testées sur une machine vierge
- ☐ Script unique pour reproduire tous les résultats
- ☐ Version de Python, CUDA, cuDNN documentée

Données :

- ☐ Données disponibles ou instructions pour les obtenir
- ☐ Prétraitement entièrement scripté (pas d'étapes manuelles)
- ☐ Splits train/val/test fixés et partagés
- ☐ Datasheet ou documentation du dataset

Expériences :

- ☐ Tous les hyperparammètres rapportés

- ☐ Graines aléatoires fixées et documentées
- ☐ Résultats moyennés sur plusieurs runs (≥ 3)
- ☐ Barres d'erreur ou intervalles de confiance rapportés
- ☐ Temps d'entraînement et ressources documentés

Modèle :

- ☐ Architecture complètement décrite
- ☐ Nombre de paramètres rapporté
- ☐ Model card incluse
- ☐ Poids du modèle disponibles (si applicable)

11.7 Comparaison des outils pour la recherche reproductible

Outil	Points forts	Limites	Cas d'usage
MLflow	Suivi d'expériences, registre de modèles	Interface parfois lourde	Projets d'équipe, production
DVC	Versionnement données, pipelines	Courbe d'apprentissage	Gros datasets, pipelines complexes
W&B	Visualisation, collaboration	SaaS (données sur le cloud)	Recherche académique, prototypage
Hydra	Configuration flexible, multi-run	Code spécifique	Balayage d'hyperparammètres
Sacred	Léger, MongoDB	Peu maintenu	Petits projets académiques
Guild AI	CLI, comparaison de runs	Communauté réduite	Expérimentation locale

11.8 Bonnes pratiques complémentaires

Règles d'or de la reproductibilité

1. **Versionner tout** : code (Git), données (DVC), environnement (`requirements.txt`), configuration (YAML).
2. **Automatiser tout** : un seul `make reproduce` doit générer tous les résultats.
3. **Documenter les choix** : pourquoi ce learning rate ? Pourquoi cette archi-

tecture ? Notez les décisions dans un journal d'expériences.

4. **Rapporter la variabilité** : toujours moyenne $\pm$ écart-type sur $n \geq 3$ runs.
5. **Conteneuriser** : un `Dockerfile` garantit un environnement identique.
6. **Archiver** : déposez code et données sur Zenodo pour obtenir un DOI pérenne.

Anti-patterns de la reproductibilité

- « Les résultats sont dans le notebook » sans script de reproduction.
- Rapporter le meilleur run sur 100 sans mentionner les 99 autres.
- Utiliser `random_state=42` partout sans justifier ni tester d'autres valeurs.
- Ne pas versionner les données de prétraitement.
- « Code disponible sur demande » (spoiler : il ne l'est jamais).
- Modifier le code après soumission sans mettre à jour le dépôt.

11.9 Mini-projet : rendre le projet de cours reproductible

Mini-projet 11 : Reproductibilité complète

Prenez le projet complet développé au fil des chapitres précédents et rendez-le entièrement reproductible :

1. **Graines aléatoires** : implémentez la fonction `set_seed()` complète et appelez-la au début de chaque script.
2. **Conteneurisation** : créez un `Dockerfile` qui installe toutes les dépendances et reproduit l'entraînement.
3. **Pipeline DVC** : définissez un `dvc.yaml` avec les étapes preprocess → train → evaluate.
4. **Documentation** :
   - Rédigez une Model Card pour votre modèle final.
   - Rédigez une Datasheet pour votre dataset.
5. **Script de reproduction** : créez un `reproduce.sh` qui clone le dépôt, construit le conteneur, et exécute le pipeline complet.
6. **Validation** : demandez à un collègue de reproduire vos résultats en suivant uniquement le README. Notez les problèmes rencontrés et corrigez-les.
7. **Archivage** : créez une release GitHub et archivez-la sur Zenodo.

reproduce.sh — Script de reproduction complète

```
#!/bin/bash
set -euo pipefail

echo "=== Reproduction complete du projet ==="

# 1. Verifier les prerequis
command -v docker >/dev/null 2>&1 || {
    echo "Docker requis. Installation: https://docs.docker.com/install/"
    exit 1
}

# 2. Construire l'image Docker
echo "[1/4] Construction de l'image Docker..."
docker build -t ml-project:reproduce .

# 3. Telecharger les donnees versionnees
echo "[2/4] Telechargement des donnees (DVC)..."
docker run --rm -v "$(pwd)/data:/app/data" ml-project:reproduce \
    dvc pull

# 4. Executer le pipeline complet
echo "[3/4] Execution du pipeline..."
```

```
docker run --rm \
    -v "$(pwd)/data:/app/data" \
    -v "$(pwd)/results:/app/results" \
    ml-project:reproduce \
    dvc repro

# 5. Verifier les resultats
echo "[4/4] Verification des metriques..."
docker run --rm \
    -v "$(pwd)/results:/app/results" \
    ml-project:reproduce \
    python src/verify_results.py \
        --expected-accuracy 0.892 \
        --tolerance 0.005

echo "=== Reproduction terminee avec succes ==="
```

11.10 Exercices

Exercice 11.1 (★ — Audit de reproductibilité)**.** Choisissez un article récent sur *Papers with Code*. Vérifiez :

1. Le code est-il disponible ?
2. Les dépendances sont-elles documentées ?
3. Les hyperparammètres sont-ils tous rapportés ?
4. Les graines aléatoires sont-elles fixées ?
5. Pouvez-vous exécuter le code avec succès ?

Rédigez un rapport d'audit de 2 pages en utilisant la checklist de la section 11.6.

Exercice 11.2 (★★ — Impact de l'aléatoire)**.** Entraînez un réseau de neurones (par exemple un CNN sur CIFAR-10) avec 10 graines aléatoires différentes. Pour chaque configuration :

1. Mesurez l'accuracy finale sur le test set.
2. Calculez la moyenne, l'écart-type, le min et le max.
3. Comparez les résultats avec et sans `torch.use_deterministic_algorithms(True)`.
4. Tracez un box plot des performances.
5. Mesurez l'impact sur le temps d'entraînement.

Exercice 11.3 (★★★ — Pipeline reproductible de bout en bout)**.** Construisez un projet ML entièrement reproductible :

1. Choisissez un problème de classification (texte ou image).

2. Implémentez le code avec la structure modulaire du chapitre 1.
3. Fixez toutes les sources d'aléatoire.
4. Créez un pipeline DVC complet.
5. Conteneurisez avec Docker.
6. Rédigez une Model Card et une Datasheet.
7. Écrivez un `reproduce.sh` qui automatise tout.
8. Faites reproduire le projet par un collègue et itérez sur les problèmes rencontrés.
9. Archivez le tout sur Zenodo et obtenez un DOI.
10. Appliquez la checklist NeurIPS et vérifiez que toutes les cases sont cochées.

11.11 Aide-mémoire

Résumé du chapitre 11

Concept	Point clé
Répétabilité	Même code, mêmes données, même environnement → mêmes résultats
Reproductibilité	Même code et données, environnement différent → mêmes résultats
Réplicabilité	Code et données différents → conclusions similaires
Graines aléatoires	Fixer Python, NumPy, PyTorch, CUDA, DataLoader
Déterminisme GPU	`torch.use_deterministic_algorithms(True)`, ralentissement de 10–20 %
Model Card	Documentation structurée du modèle (architecture, biais, limites)
Datasheet	Documentation structurée du dataset (collecte, composition, licence)
Archivage	Zenodo (DOI), Hugging Face Hub, Papers with Code
Checklist	Code, données, hyperparammètres, variabilité, ressources
Automatisation	`reproduce.sh` + Docker + DVC pour reproduction complète

Annexe A

Annexe : Comparaison des Outils MLOps

Ce chapitre fournit une référence complète des outils utilisés dans l'écosystème MLOps, organisés par catégorie. Pour chaque outil, nous indiquons ses forces, ses faiblesses et son cas d'usage idéal.

A.1 Gestion d'environnements

Outil	Description	Forces	Faiblesses	Cas idéal
venv	Module standard Python pour environnements virtuels	Intégré, simple, léger, aucune installation requise	Pas de dépendances système, pas de verrouillage natif	Petits projets Python purs
Conda	Gestionnaire multi-langage avec dépendances système	Gère CUDA/cuDNN, multi-OS, écosystème riche	Lent, environnements volumineux, conflits conda/pip	Projets ML avec GPU
Poetry	Gestion moderne avec verrouillage et publication	Verrouillage automatique, `pyproject.toml`, publication PyPI	Pas de dépendances système, parfois lent	Bibliothèques Python, projets d'équipe
uv	Remplacement pip ultra-rapide écrit en Rust	10–100× plus rapide que pip, compatible pip	Jeune, pas de dépendances système	Tout projet Python (remplacement pip)
pixi	Gestionnaire Conda rapide écrit en Rust	Rapide, fichier de lock, multi-plateforme, dépendances système	Jeune, écosystème encore limité	Projets multi-langage, remplacement Conda

A.2 Contrôle de version

Outil	Description	Forces	Faiblesses	Cas idéal
Git	Système de contrôle de version distribué	Standard universel, branches, merge, écosystème immense	Mauvais pour les gros fichiers binaires	Tout code source
DVC	Versionnement de données et pipelines ML	Compatible Git, pipelines reproductibles, stockage distant (S3, GCS)	Courbe d'apprentissage, overhead pour petits projets	Gros datasets, pipelines ML
Git LFS	Extension Git pour les gros fichiers	Intégré à Git, transparent, support GitHub/GitLab	Bande passante limitée, coût de stockage	Fichiers binaires (< quelques Go)
LakeFS	Contrôle de version pour data lakes	Compatible S3, branches sur les données, CI/CD pour données	Infrastructure lourde, complexe pour petits projets	Data lakes en production

A.3 Suivi d'expériences

Outil	Description	Forces	Faiblesses	Cas idéal
MLflow	Plateforme open-source de gestion du cycle ML	Open-source, registre de modèles, API simple, déploiement intégré	Interface basique, scalabilité limitée en natif	Équipes, production, on-premise
W&B	Plateforme SaaS de suivi et visualisation	Visualisations riches, sweeps, rapports collaboratifs	SaaS (données sur le cloud), coût pour grandes équipes	Recherche, prototypage rapide
Neptune	Plateforme de suivi avec métadonnées flexibles	Interface fluide, métadonnées custom, intégrations nombreuses	SaaS, coût élevé pour gros volumes	Équipes de recherche appliquée
ClearML	Plateforme MLOps complète open-source	Gratuit (self-hosted), orchestration intégrée, auto-logging	Complexe à configurer, documentation inégale	Startups, PME, self-hosted
Comet	Suivi d'expériences avec comparaison visuelle	Diff de code automatique, panels personnalisés	SaaS, fonctionnalités avancées payantes	Équipes de recherche

A.4 Orchestration de pipelines

Outil	Description	Forces	Faiblesses	Cas idéal
Airflow	Orchestrateur de workflows (Apache)	Standard industrie, vaste écosystème, planification cron, UI web	Lourd, DAGs statiques, pas conçu pour le ML	Pipelines de données en production
Prefect	Orchestration moderne Python-native	API Pythonique, UI élégante, gestion d'échecs, cloud hybrid	Moins mature qu'Airflow, écosystème plus petit	Pipelines ML modernes
Dagster	Orchestrateur orienté données (data-aware)	Typage des assets, tests intégrés, observabilité	Courbe d'apprentissage, communauté plus réduite	Pipelines data-centric
Luigi	Orchestrateur de tâches (Spotify)	Simple, dépendances entre tâches, idéal pour batch	Pas de planification native, UI limitée	Pipelines batch simples
Kubeflow	Pipelines ML sur Kubernetes	Scalable, intégration K8s, GPU natif, composants ML	Nécessite Kubernetes, complexe à opérer	ML à grande échelle sur K8s

A.5 Conteneurisation

Outil	Description	Forces	Faiblesses	Cas idéal
Docker	Standard de conteneurisation	Ubiquitaire, écosystème immense, Docker Compose, GPU (nvidia-docker)	Démon root par défaut, images volumineuses	Tout projet en production
Podman	Alternative rootless à Docker	Sans démon, rootless, compatible CLI Docker	Moins de tooling, Compose expérimental	Environnements sécurisés
Singularity / Apptainer	Conteneurs pour le HPC	Pas de root requis, compatible clusters HPC, supporte GPU	Moins répandu hors HPC, écosystème plus petit	Clusters universitaires, HPC

A.6 Déploiement et serving

Outil	Description	Forces	Faiblesses	Cas idéal
FastAPI	Framework web Python asynchrone	Rapide, typage Pydantic, doc auto (Swagger), asynchrone	Pas spécifique ML, batching manuel	APIs ML légères, prototypage
TorchServe	Serveur de modèles PyTorch	Optimisé PyTorch, batching, multi-modèles, métriques	PyTorch uniquement, configuration verbose	Modèles PyTorch en production
Triton	Serveur d'inférence NVIDIA	Multi-framework, GPU optimisé, batching dynamique, ensembles	Complexe, nécessite GPU NVIDIA	Inférence GPU haute performance
BentoML	Plateforme de packaging et serving ML	Multi-framework, packaging simple, Bentofile, GPU support	Abstraction supplémentaire, debugging parfois difficile	Standardisation du déploiement
Seldon Core	Déploiement ML sur Kubernetes	A/B testing, canary, monitoring, multi-framework	Nécessite Kubernetes, complexe	ML à grande échelle sur K8s

A.7 CI/CD

Outil	Description	Forces	Faiblesses	Cas idéal
GitHub Actions	CI/CD intégré à GitHub	Intégration native GitHub, marketplace riche, runners gratuits	Limité à GitHub, runners GPU payants	Projets open-source sur GitHub
GitLab CI	CI/CD intégré à GitLab	Registre Docker intégré, runners self-hosted, pipeline as code	Plus complexe, UI moins intuitive	Entreprises self-hosted
Jenkins	Serveur d'automatisation open-source	Extrêmement flexible, plugins, self-hosted, gratuit	Interface vieillissante, maintenance lourde	Grandes entreprises, pipelines custom
CircleCI	Plateforme CI/CD cloud	Configuration YAML simple, caching avancé, parallélisme	Coût élevé, runners GPU limités	Startups, projets moyens

A.8 Monitoring et observabilité

Outil	Description	Forces	Faiblesses	Cas idéal
Evidently	Monitoring open-source de modèles ML	Open-source, rapports HTML, détection de drift, intégration Grafana	Dashboards temps réel limités en open-source	Monitoring de drift, rapports batch
WhyLabs	Observabilité ML en temps réel	Profiling automatique, alertes, intégration simple (whylogs)	SaaS, coût pour gros volumes	Monitoring en production, temps réel
Fiddler	Plateforme d'explicabilité et monitoring	Explicabilité intégrée, détection de biais, alertes	SaaS coûteux, setup initial complexe	Modèles réglementés (finance, santé)
NannyML	Détection de drift sans ground truth	Estimation de performance sans labels, open-source	Spécialisé drift, moins polyvalent	Monitoring quand les labels sont retardés

A.9 Synthèse et recommandations

Choisir les bons outils selon le contexte

Projet solo / TP de cours venv + pip, Git, MLflow (local), FastAPI, GitHub Actions.

Équipe de recherche académique Conda ou pixi, Git + DVC, W&B ou MLflow, Singularity (HPC), GitHub Actions.

Startup / PME Poetry ou uv, Git + DVC, MLflow ou ClearML (self-hosted), Docker, Prefect, FastAPI ou BentoML, GitHub Actions, Evidently.

Grande entreprise Conda ou pixi, Git + DVC + LakeFS, MLflow (serveur), Docker + Kubernetes, Kubeflow ou Dagster, Triton ou Seldon, GitLab CI ou Jenkins, WhyLabs ou Fiddler.

Remarque A.1. L'écosystème MLOps évolue très rapidement. Les outils mentionnés ici reflètent l'état de l'art en 2025. Avant d'adopter un outil, vérifiez :

- la fréquence des mises à jour (dernière release),
- la taille de la communauté (stars GitHub, forum),
- la compatibilité avec votre infrastructure existante,
- le coût total (licences, infrastructure, formation).

Bibliographie

[1] C. Huyen. *Designing Machine Learning Systems*. O'Reilly Media, 2022.

[2] M. Kleppmann. *Designing Data-Intensive Applications*. O'Reilly Media, 2017.

[3] N. Gift, A. Deza. *Practical MLOps*. O'Reilly Media, 2021.

[4] M. Treveil et al. *Introducing MLOps*. O'Reilly Media, 2020.

[5] MLflow Documentation. https://mlflow.org/docs/latest/index.html, 2024.

[6] DVC Documentation. https://dvc.org/doc, 2024.

[7] Weights & Biases Documentation. https://docs.wandb.ai/, 2024.

[8] Docker Documentation. https://docs.docker.com/, 2024.

[9] FastAPI Documentation. https://fastapi.tiangolo.com/, 2024.

[10] D. Sculley et al. Hidden Technical Debt in Machine Learning Systems. *NeurIPS*, 2015.

[11] A. Paleyes, R. Urma, N. Lawrence. Challenges in Deploying Machine Learning : a Survey of Case Studies. *ACM Computing Surveys*, 2022.

[12] J. Pineau et al. Improving Reproducibility in Machine Learning Research. *JMLR*, 22(164) :1–20, 2021.